Theoretische Informatik

Beweise 101

Nicolas Wehrli, ETH Zurich

December 29, 2023

Disclaimer

This is a personal summary I completed during my time as a Teaching Assistant for this course at ETH Zurich. It is not an official document and the content is neither **complete** nor does it **guarantee** to be correct.

I tried to give examples to accompany the theory and show some common recipes to solve certain exercises.

If you find any mistakes or inconsistencies, I would be very happy about a quick message so that I can correct them;)

You can reach me under nwehrl@student.ethz.ch and the LaTex Source code can be found here.

The corresponding slides from my exercise session are available on a small website of mine.

Note that the summary does not cover 'Grammatiken' since this was only covered after the second midterm in the autumn semester of 2023. The non-deterministic hierarchical stuff (section 6.4 in the german version of the book) was also not covered since the endterm preparation was prioritised.

These parts may be added later.

Hope this helps. Good luck in your future and may you achieve your goals!

Feel free to reach out if you have any questions.

Contents

1 Grundbegriffe

Für eine Menge A bezeichnet $|A|$ die Kardinalität von A und $\mathcal{P}(A) = \{S \mid S \subseteq A\}$ die Potenzmenge von A.

In diesem Kurs definieren wir $\mathbb{N} = \{0, 1, 2, \dots\}$.

1.1 Alphabet

Definition Alphabet

Eine endliche, nichtleere Menge Σ heisst **Alphabet**. Die Elemente eines Alphabets werden **Buchstaben (Zeichen, Symbole)** genannt.

Beispiele

- $\Sigma_{\text{bool}} = \{0, 1\}$
- $\Sigma_{\text{lat}} = \{a, ..., z\}$
- $\Sigma_{\text{Tastatur}} = \Sigma_{\text{lat}} \cup \{A, ..., Z, \text{␣}, >, <, (,), ..., !\}$
- $\Sigma_{\text{logic}} = \{0, 1, (,), \wedge, \vee, \neg\}$
- $\Sigma_{abc} = \{a, b, c\}$ (**unser Beispiel für weitere Definitionen**)

1.2 Wort

Definition Wort

- Sei Σ ein Alphabet. Ein **Wort** über Σ ist eine endliche (eventuell leere) Folge von Buchstaben aus Σ.
- Das **leere Wort** λ ist die leere Buchstabenfolge.
- Die **Länge** $|w|$ eines Wortes w ist die Länge des Wortes als Folge, i.e. die Anzahl der Vorkommen von Buchstaben in w.
- Σ^* ist die Menge aller Wörter über Σ. $\Sigma^+ := \Sigma^* \setminus \{\lambda\}$ ist Menge aller nichtleeren Wörter über Σ.
- Seien $x \in \Sigma^*$ und $a \in \Sigma$. Dann ist $|x|_a$ definiert als die Anzahl der Vorkommen von a in x.

Achtung Metavariablen! I.e. Das a in der Definition ist steht für einen beliebigen Buchstaben aus Σ und **nicht** nur für den Buchstaben 'a', der in Σ sein könnte.

Bemerkungen

- Wir schreiben Wörter ohne Komma, i.e. eine Folge $x_1, x_2, ..., x_n$ schreiben wir $x_1x_2...x_n$.
- $|\lambda| = 0$ aber $|\text{␣}| = 1$ von Σ_{Tastatur}.
- Der Begriff **Wort** als Fachbegriff der Informatik entspricht **nicht** der Bedeutung des Begriffs Wort in natürlichen Sprachen!
- E.g. Mit ␣ kann der Inhalt eines Buches oder ein Programm als ein Wort über Σ_{Tastatur} betrachtet werden.

Beispiel Verschiedene Wörter über Σ_{abc}:

a, aa, aba, cba, $caaaab$ etc.

Die **Verkettung (Konkatenation)** für ein Alphabet Σ ist eine Abbildung Kon: $\Sigma^* \times \Sigma^* \to \Sigma^*$, so dass

$$\text{Kon}(x, y) = x \cdot y = xy$$

für alle $x, y \in \Sigma^*$.

- Die Verkettung Kon (i.e. Kon von einem Kon (über das gleiche Alphabet Σ)) ist eine assoziative Operation über Σ^*.

$$\text{Kon}(u, \text{Kon}(v, w)) = \text{Kon}(\text{Kon}(u, v), w), \ \forall u, v, w \in \Sigma^*$$

- $x \cdot \lambda = \lambda \cdot x = x, \ \forall x \in \Sigma^*$
- $\implies$ (Σ^*, Kon) ist ein Monoid mit neutralem Element λ.
- Kon nur kommutativ, falls $|\Sigma| = 1$.
- $|xy| = |x \cdot y| = |x| + |y|$. (Wir schreiben ab jetzt xy statt Kon(x, y))

Beispiel

Wir betrachten wieder Σ_{abc}. Sei $x = abba$, $y = cbcbc$, $z = aaac$.

- $\text{Kon}(x, \text{Kon}(y, z)) = \text{Kon}(x, yz) = xyz = abbacbcbcaaac$
- $|xy| = |abbacbcbc| = 9 = 4 + 5 = |abba| + |cbcbc| = |x| + |y|$

Für eine Wort $a = a_1 a_2 ... a_n$, wobei $\forall i \in \{1, 2, ..., n\}$. $a_i \in \Sigma$, bezeichnet $a^{\text{R}} = a_n a_{n-1} ... a_1$ die **Umkehrung (Reversal)** von a.

Sei Σ ein Alphabet. Für alle $x \in \Sigma^*$ und alle $i \in \mathbb{N}$ definieren wir die i-te **Iteration** x^i von x als

$$x^0 = \lambda, x^1 = x \text{ und } x^i = xx^{i-1}.$$

Beispiel

Wir betrachten wieder Σ_{abc}. Sei $x = abba$, $y = cbcbc$, $z = aaac$.

- $z^{\text{R}} = (aaac)^{\text{R}} = caaa$
- $x^{\text{R}} = (abba)^{\text{R}} = abba$
- $x^0 = \lambda$
- $y^2 = yy^{2-1} = yy = cbcbccbcbc$
- $z^3 = zz^2 = zzz = aaacaaacaaac$
- $(x^{\text{R}} z^{\text{R}})^{\text{R}} = ((abba)^{\text{R}}(aaac)^{\text{R}})^{\text{R}} = (abbacaaa)^{\text{R}} = aaacabba$

Seien $v, w \in \Sigma^*$ für ein Alphabet Σ.

- v heisst ein **Teilwort** von $w \iff \exists x, y \in \Sigma^* : \ w = xvy$
- v heisst ein **Präfix** von $w \iff \exists y \in \Sigma^* : \ w = vy$
- v heisst ein **Suffix** von $w \iff \exists x \in \Sigma^* : \ w = xv$
- $v \neq \lambda$ heisst ein **echtes** Teilwort (Präfix, Suffix) von $w \iff v \neq w$ und v Teilwort(Präfix, Suffix) von w

Beispiel

Wir betrachten wieder Σ_{abc}. Sei $x = abba$, $y = cbcbc$, $z = aaac$.

- bc ist ein echtes Suffix von y
- $abba$ ist kein echtes Teilwort von x.
- $cbcb$ ist ein echtes Teilwort und echtes Präfix von y.
- ac ist ein echtes Suffix.
- $abba$ ist ein Suffix, Präfix und Teilwort von x.

Aufgabe 1

Sei Σ ein Alphabet und sei $w \in \Sigma^*$ ein Wort der Länge $n \in \mathbb{N} \setminus \{0\}$. Wie viele unterschiedliche Teilwörter kann w höchstens haben?

Lösung

Wir haben $w = w_1w_2...w_n$ mit $w_i \in \Sigma$ für $i = 1, ..., n$. Wie viele Teilwörter beginnen mit w_1? Wie viele Teilwörter beginnen mit w_2?

Wir haben also $n + (n-1) + ... + 1 = \frac{n(n+1)}{2}$ Teilwörter. Etwas fehlt aber in unserer Berechnung...

Das leere Wort λ ist auch ein Teilwort! Also haben wir $\frac{n(n+1)}{2} + 1$ Teilwörter.

Aufgabe 2

Sei $\Sigma = \{a, b, c\}$ und $n \in \mathbb{N}$. Bestimme die Anzahl der Wörter aus Σ^n, die das Teilwort a enthalten.

Lösung

In solchen Aufgaben ist es manchmal einfach, das Gegenteil zu berechnen und so auf die Lösung zu kommen. Wie viele Wörter aus Σ^n enthalten das Teilwort a **nicht**?

Da wir jetzt die Anzahl Wörter der Länge n wollen, die nur b und c enthalten, kommen wir auf $|\{b, c\}|^n = 2^n$.

Daraus folgt, dass genau $|\Sigma|^n - 2^n = 3^n - 2^n$ Wörter das Teilwort a enthalten.

Aufgabe 3

Sei $\Sigma = \{a, b, c\}$ und $n \in \mathbb{N} \setminus \{0\}$. Bestimme die Anzahl der Wörter aus Σ^n, die das Teilwort aa nicht enthalten.

Lösung

Wir bezeichnen die Menge aller Wörter mit Länge n über Σ, die aa nicht enthalten als L_n.

Schauen wir mal die ersten zwei Fälle an:

- $L_1 = \{a, b, c\} \implies |L_1| = 3$
- $L_2 = \{ab, ac, ba, bb, bc, ca, cb, cc\} \implies |L_2| = 8$

Nun können wir für $m \geq 3$ jedes Wort $w \in L_m$ als Konkatination $w = x \cdot y \cdot z$ schreiben, wobei wir zwei Fälle unterscheiden:

(a) $\mathbf{z \neq a}$

In diesem Fall kann $y \in \{a, b, c\}$ sein, ohne dass die Teilfolge aa entsteht und somit ist xy ein beliebiges Wort aus L_{m-1}.

Dann könnten wir alle Wörter in diesem Case durch $L_{m-1} \cdot \{b, c\}$ beschreiben, was uns die Kardinalität $2 \cdot |L_{m-1}|$ gibt.

(b) $\mathbf{z = a}$

In diesem Fall muss $y \neq a$ sein, da sonst aa entstehen würde.

Somit kann xy nur in b oder c enden. x kann aber ein beliebiges Wort der Länge $m-2$ sein.

Deshalb können wir alle Wörter in diesem Case durch $L_{m-2} \cdot \{b, c\} \cdot \{a\}$ beschreiben. Kardinalität: $2 \cdot |L_{m-2}|$.

Daraus folgt

$$|L_n| = \begin{cases} 3 & n = 1 \\ 8 & n = 2 \\ 2|L_{n-1}| + 2|L_{n-2}| & n \geq 3 \end{cases}$$

Sei $\Sigma = \{s_1, s_2, ..., s_m\}, m \geq 1$, ein Alphabet und sei $s_1 < s_2 < ... < s_m$ eine Ordnung auf Σ. Wir definieren die **kanonische Ordnung** auf Σ^* für $u, v \in \Sigma^*$ wie folgt:

$$u < v \iff |u| < |v| \vee (|u| = |v| \wedge u = x \cdot s_i \cdot u' \wedge x \cdot s_j \cdot v')$$
$$\text{für irgendwelche } x, u', v' \in \Sigma^* \text{ und } i < j.$$

Sei $\Sigma_{abc} = \{a, b, c\}$ und wir betrachten folgende Ordnung auf Σ_{abc}: $c < a < b$.

Was wäre die kanonische Ordnung folgender Wörter?

c, *abc*, *aaac*, *aaab*, *bacc*, *a*, λ

λ, *c*, *a*, *abc*, *aaac*, *aaab*, *bacc*

1.3 Sprache

Eine **Sprache** L über einem Alphabet Σ ist eine Teilmenge von Σ^*.

- Das Komplement $L^{\complement}$ der Sprache L bezüglich Σ ist die Sprache $\Sigma^* \setminus L$.
- $L_\emptyset = \emptyset$ ist die **leere Sprache**.
- $L_\lambda = \{\lambda\}$ ist die einelementige Sprache, die nur aus dem leeren Wort besteht.

Konkatenation von Sprachen

Sind L_1 und L_2 Sprachen über Σ, so ist

$$L_1 \cdot L_2 = L_1 L_2 = \{vw \mid v \in L_1 \text{ und } w \in L_2\}$$

die **Konkatenation** von L_1 und L_2.

Ist L eine Sprache über Σ, so definieren wir

$$L^0 := L_\lambda \text{ und } L^{i+1} := L^i \cdot L \text{ für alle } i \in \mathbb{N},$$
$$L^* = \bigcup_{i \in \mathbb{N}} L^i \text{ und } L^+ = \bigcup_{i \in \mathbb{N} \setminus \{0\}} L^i = L \cdot L^*.$$

L^* nennt man den **Kleene'schen Stern** von L.

Man bemerke, dass $\Sigma^i = \{x \in \Sigma^* \mid |x| = i\}$, $L_\emptyset L = L_\emptyset = \emptyset$ und $L_\lambda \cdot L = L$.

Mögliche Sprachen über Σ_{abc}

- $L_1 = \emptyset$
- $L_2 = \{\lambda\}$
- $L_3 = \{\lambda, ab, baca\}$
- $L_4 = \Sigma^*_{abc}$, $L_5 = \Sigma^+_{abc}$, $L_6 = \Sigma_{abc}$ oder $L_7 = \Sigma^{27}_{abc}$
- $L_8 = \{c\}^* = \{c^i \mid i \in \mathbb{N}\}$
- $L_9 = \{a^p \mid p \text{ ist prim.}\}$
- $L_{10} = \{c^i a^{3i^2} b a^i c \mid i \in \mathbb{N}\}$

λ ist ein Wort über jedes Alphabet. Aber es muss nicht in jeder Sprache enthalten sein!

Seien L_1, L_2 und L_3 Sprachen über einem Alphabet Σ. Dann gilt

$$L_1L_2 \cup L_1L_3 = L_1(L_2 \cup L_3) \tag{1}$$
$$L_1(L_2 \cap L_3) \subseteq L_1L_2 \cap L_1L_3 \tag{2}$$

Weshalb nicht '=' bei (2)?

Sei $\Sigma = \Sigma_{\text{bool}} = \{0, 1\}$, $L_1 = \{\lambda, 1\}$, $L_2 = \{0\}$ und $L_3 = \{10\}$.

Dann haben wir $L_1(L_2 \cap L_3) = \emptyset \neq \{10\} = L_1L_2 \cap L_1L_3$.

Beweise im Buch/Vorlesung

Homomorphismus

Seien Σ_1 und Σ_2 zwei beliebige Alphabete. Ein Homomorphismus von Σ_1^* nach Σ_2^* ist jede Funktion $h : \Sigma_1^* \to \Sigma_2^*$ mit den folgenden Eigenschaften:

(i) $h(\lambda) = \lambda$ und

(ii) $h(uv) = h(u) \cdot h(v)$ für alle $u, v \in \Sigma_1^*$.

Wir können Probleme etc. in anderen Alphabeten kodieren. So wie wir verschiedenste Konzepte, die wir auf Computer übertragen in Σ_{bool} kodieren.

2 Algorithmische Probleme

Mathematische Definition folgt in Kapitel 4 (Turingmaschinen).

Algorithmen - Provisorische Definition

Vorerst betrachten wir Programme, die **für jede zulässige Eingabe halten und eine Ausgabe liefern**, als Algorithmen.
Wir betrachten ein Programm (Algorithmus) A als Abbildung $A : \Sigma_1^* \to \Sigma_2^*$ für beliebige Alphabete Σ_1 und Σ_2. Dies bedeutet, dass

(i) die Eingaben als Wörter über Σ_1 kodiert sind,

(ii) die Ausgaben als Wörter über Σ_2 kodiert sind und

(iii) A für jede Eingabe eine eindeutige Ausgabe bestimmt.

A und B äquivalent $\iff$ Eingabealphabet Σ gleich, $A(x) = B(x), \forall x \in \Sigma^*$

Ie. diese Notion von "Äquivalenz" bezieht sich nur auf die Ein und Ausgabe.

Entscheidungsproblem

Das **Entscheidungsproblem** (Σ, L) für ein gegebenes Alphabet Σ und eine gegebene Sprache $L \subseteq \Sigma^*$ ist, für jedes $x \in \Sigma^*$ zu entscheiden, ob

$$x \in L \text{ oder } x \notin L.$$

Ein Algorithmus A **löst** das Entscheidungsproblem (Σ, L), falls für alle $x \in \Sigma^*$ gilt:

$$A(x) = \begin{cases} 1, & \text{falls } x \in L, \\ 0, & \text{falls } x \notin L. \end{cases}$$

Wir sagen auch, dass A die Sprache L erkennt.

Rekursive Sprachen

Wenn für eine Sprache L ein Algorithmus existiert, der L erkennt, sagen wir, dass L **rekursiv** ist.

Wir sind oft an spezifischen Eigenschaften von Wörtern aus Σ^* interessiert, die wir mit einer Sprache $L \subseteq \Sigma^*$ beschreiben können.

Dabei sind dann L die Wörter, die die Eigenschaft haben und $L^{\complement} = \Sigma^* \setminus L$ die Wörter, die diese Eigenschaft nicht haben.

Jetzt ist die allgemeine Formulierung von Vorteil!

i. **Primzahlen finden**:

   Entscheidungsproblem $(\Sigma_{\text{bool}}, L_p)$ wobei
   $L_p = \{x \in (\Sigma_{\text{bool}})^* \mid \text{Nummer}(x) \text{ ist prim}\}$.

ii. **Syntaktisch korrekte Programme**:

   Entscheidungsproblem $(\Sigma_{\text{Tastatur}}, L_{C++})$ wobei
   $L_{C++} = \{x \in (\Sigma_{\text{Tastatur}})^* \mid x \text{ ist ein syntaktisch korrektes C++ Programm}\}$.

iii. **Hamiltonkreise finden:**

   Entscheidungsproblem (Σ, HK) wobei $\Sigma = \{0, 1, \#\}$ und
   HK= $\{x \in \Sigma^* \mid x \text{ kodiert einen Graphen, der einen Hamiltonkreis enthält.}\}$

Äquivalenzprobleme $\subset$ Entscheidungsprobleme

Seien Σ und Γ zwei Alphabete.

- Wir sagen, dass ein Algorithmus A eine **Funktion (Transformation)** $f : \Sigma^* \to \Gamma^*$ **berechnet (realisiert)**, falls
$$A(x) = f(x) \text{ für alle } x \in \Sigma^*$$
- Sei $R \subseteq \Sigma^* \times \Gamma^*$ eine Relation in Σ^* und Γ^*. Ein Algorithmus A **berechnet** R (bzw. **löst das Relationsproblem** R), falls für jedes $x \in \Sigma^*$, für das ein $y \in \Gamma^*$ mit $(x, y) \in R$ existiert, gilt:
$$(x, A(x)) \in R$$

Optimierungsproblem

Ein **Optimierungsproblem** ist ein 6-Tupel $\mathcal{U} = (\Sigma_I, \Sigma_O, L, M, \text{cost}, \text{goal})$, wobei:

(i) Σ_I ist ein Alphabet (genannt **Eingabealphabet**),

(ii) Σ_O ist ein Alphabet (genannt **Ausgabealphabet**),

(iii) $L \subseteq \Sigma_I^*$ ist die Sprache der **zulässigen Eingaben** (als Eingaben kommen nur Wörter in Frage, die eine sinnvolle Bedeutung haben). Ein $x \in L$ wird ein **Problemfall (Instanz) von** $\mathcal{U}$ genannt.

(iv) M ist eine Funktion von L nach $\mathcal{P}(\Sigma_O^*)$, und für jedes $x \in L$ ist $M(x)$ die **Menge der zulässigen Lösungen für** x,

(v) **cost** ist eine Funktion, **cost**: $\bigcup_{x \in L}(\mathcal{M}(x) \times \{x\}) \to \mathbb{R}^+$, genannt **Kostenfunktion**,

(vi) **goal** $\in \{\text{Minimum}, \text{Maximum}\}$ ist das **Optimierungsziel**.

Eine zulässige Lösung $\alpha \in \mathcal{M}(x)$ heisst **optimal** für den Problemfall x des Optimierungsproblems $\mathcal{U}$, falls

$$\text{cost}(\alpha, x) = \mathbf{Opt}_{\mathcal{U}}(x) = \text{ goal}\{\text{cost}(\beta, x) \mid \beta \mathcal{M}(x)\}.$$

Ein Algorithmus A **löst** $\mathcal{U}$, falls für jedes $x \in L$

(i) $A(x) \in \mathcal{M}(x)$

(ii) $\text{cost}(A(x), x) = \text{ goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}$.

3 Kolmogorov Komplexität

3.1 Theorie

Algorithmen generieren Wörter

Sei Σ ein Alphabet und $x \in \Sigma^*$. Wir sagen, dass ein Algorithmus A das Wort x **generiert**, falls A für die Eingabe λ die Ausgabe x liefert.

Beispiel:

```
A_n:  begin
          for i = 1 to n;
              write(01);
      end
```

A_n generiert $(01)^n$.

Aufzählungsalgorithmus

Sei Σ ein Alphabet und sei $L \subseteq \Sigma^*$. A ist ein **Aufzählungsalgorithmus für** L, falls A für jede Eingabe $n \in \mathbb{N} \setminus \{0\}$ die Wortfolge $x_1, ..., x_n$ ausgibt, wobei $x_1, ..., x_n$ die kanonisch n ersten Wörter in L sind.

Aufgabe 2.21

Beweisen Sie, dass eine Sprache L genau dann rekursiv ist, wenn ein Aufzählungsalgorithmus für L existiert.

Das **Entscheidungsproblem** (Σ, L) für ein gegebenes Alphabet Σ und eine gegebene Sprache $L \subseteq \Sigma^*$ ist, für jedes $x \in \Sigma^*$ zu entscheiden, ob

$$x \in L \text{ oder } x \notin L.$$

Ein Algorithmus A **löst** das Entscheidungsproblem (Σ, L), falls für alle $x \in \Sigma^*$ gilt:

$$A(x) = \begin{cases} 1, & \text{falls } x \in L, \\ 0, & \text{falls } x \notin L. \end{cases}$$

Wir sagen auch, dass A die Sprache L erkennt.

L rekursiv ($\Longrightarrow$) existiert Aufzählungsalgorithmus:

Sei A ein Algorithmus, der L erkennt. Wir beschreiben nun einen Aufzählungsalgorithmus B konstruktiv.

Algorithm 1 $B(\Sigma, n)$

```
i ← 0
while i ≤ n do
    w ← kanonisch nächstes Wort über Σ*
    if A(w) = 1 then
        print(w)
        i ← i + 1
    end if
end while
```

Aufzählungsalgorithmus $B \implies L$ rekursiv:

Algorithm 2 $A(\Sigma, w)$

```
n ← |Σ|^(|w|+1)
L ← B(Σ, n)
if w ∈ L then
    print(1)
else
    print(0)
end if
```

Es gibt ein kleines Problem. B könnte unendlich lange laufen, falls $n > |L|$.
Es sollte nicht so schwierig sein, B zu modifizieren, dass es die Berechnung aufhört, falls es keine weiteren Wörter in L gibt.

Information messen

Wir beschränken uns auf Σ_{bool}

Kolmogorov-Komplexität

Für jedes Wort $x \in (\Sigma_{\text{bool}})^*$ ist die **Kolmogorov-Komplexität $K(x)$ des Wortes** x das Minimum der binären Längen, der Pascal-Programme, die x generieren.

$K(x)$ ist die kürzestmögliche Länge einer Beschreibung von x.

Die einfachste (und triviale) Beschreibung von x, ist wenn man x direkt angibt.

x kann aber eine Struktur oder Regelmässigkeit haben, die eine Komprimierung erlaubt.

Welche Programmiersprache gewählt wird verändert die Kolmogorov-Komplexität nur um eine Konstante. (Satz 2.1)

Beispiel

Sei $w = 01010101010101010101010101010101010101$. Die Länge von w ist $|w| = 40$ und die triviale Beschreibunglänge wäre wie gegeben 40.

Aber durch die Regelmässigkeit von einer 20-fachen Wiederholung der Sequenz 01, können w auch durch $(01)^{20}$ beschreiben. Hierbei ist die Beschreibungslänge ein wenig mehr als 4 Zeichen.

Grundlegende Resultate

Es existiert eine Konstante d, so dass für jedes $x \in (\Sigma_{\text{bool}})^*$

$$K(x) \leq |x| + d$$

Die **Kolmogorov-Komplexität einer natürlichen Zahl** n ist $K(n) = K(\text{Bin}(n))$.

Lemma 2.5 - Nichtkomprimierbar

Für jede Zahl $n \in \mathbb{N} \setminus \{0\}$ existiert ein Wort $w_n \in (\Sigma_{\text{bool}})^n$, so dass

$$K(w_n) \geq |w_n| = n$$

Beweis

Es gibt 2^n Wörter $x_1, ..., x_{2^n}$ über Σ_{bool} der Länge n. Wir bezeichnen $C(x_i)$ als den Bitstring des kürzesten Programms, der x_i generieren kann. Es ist klar, dass für $i \neq j : C(x_i) \neq C(x_j)$.

Die Anzahl der nichtleeren Bitstrings, i.e. der Wörter der Länge $< n$ über Σ_{bool} ist:

$$\sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n$$

Also muss es unter den Wörtern $x_1, ..., x_{2^n}$ mindestens ein Wort x_k mit $K(x_k) \geq n$ geben.

■

Satz 2.1 - Programmiersprachen

Für jedes Wort $x \in (\Sigma_{\text{bool}})^*$ und jede Programmiersprache A sei $K_A(x)$ die Kolmogorov-Komplexität von x bezüglich der Programmiersprache A.

Seien A und B Programmiersprachen. Es existiert eine Konstante $c_{A,B}$, die nur von A und B abhängt, so dass

$$|K_A(x) - K_B(x)| \leq c_{A,B}$$

für alle $x \in (\Sigma_{\text{bool}})^*$.

Beweis im Buch/Vorlesung

Ein zufälliges Wort

Ein Wort $x \in (\Sigma_{\text{bool}})^*$ heisst **zufällig**, falls $K(x) \geq |x|$.
Eine Zahl n heisst **zufällig**, falls $K(n) = K(\text{Bin}(n)) \geq \lceil \log_2(n+1) \rceil - 1$.

Jede Binär-Darstellung beginnt immer mit einer 1, deshalb können wir die Länge der Binär-Darstellung um 1 verkürzen.

Zufälligkeit hier bedeutet, dass ein Wort völlig unstrukturiert ist und sich nicht komprimieren lässt. Es hat nichts mit Wahrscheinlichkeit zu tun.

Satz 2.2

Sei L eine Sprache über Σ_{bool}. Sei für jedes $n \in \mathbb{N} \setminus \{0\}$, z_n das n-te Wort in L bezüglich der kanonischen Ordnung. Wenn ein Programm A_L existiert, das das Entscheidungsproblem $(\Sigma_{\text{bool}}, L)$ löst, dann gilt für alle $n \in \mathbb{N} \setminus \{0\}$, dass

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + c$$

wobei c eine von n unabhängige Konstante ist.

Beweisidee

Wir können aus A_L, ein Programm entwerfen, dass das kanonisch n-te Wort generiert, indem wir in der kanonischen Reihenfolge alle Wörter $x \in (\Sigma_{bool})^*$ durchgehen und mit A_L entscheiden, ob $x \in L$. Dann können wir einen Counter c haben und den Prozess abbrechen, wenn der Counter $c = n$ wird und dann dieses Wort ausgeben.

Wir sehen, dass dieses Programm ausser der Eingabe n immer gleich ist. Sei die Länge dieses Programms c, dann können wir für das n-te Wort der Sprache L, z_n, die Kolmogorov-Komplexität auf n reduzieren, bzw:

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + c$$

■

Primzahlsatz

Für jede positive ganz Zahl n sei $\text{Prim}(n)$ die Anzahl der Primzahlen kleiner gleich n.

$$\lim_{n \to \infty} \frac{\text{Prim}(n)}{n/\ln n} = 1$$

Nützliche Ungleichung

$$\ln n - \frac{3}{2} < \frac{n}{\text{Prim(n)}} < \ln n - \frac{1}{2}$$

für alle $n \geq 67$.

Lemma 2.6 - schwache Version des Primzahlsatzes

Sei $n_1, n_2, n_3, ...$ eine steigende unendliche Folge natürlicher Zahlen mit $K(n_i) \geq \lceil \log_2 n_i \rceil / 2$. Für jedes $i \in \mathbb{N} \setminus \{0\}$ sei q_i die grösste Primzahl, die die Zahl n_i teilt. Dann ist die Menge $Q = \{q_i \mid i \in \mathbb{N} \setminus \{0\}\}$ unendlich.

Beweis: Wir beweisen diese Aussage per Widerspruch:

Nehmen wir zum Widerspruch an, dass die Menge $Q = \{q_i \mid i \in \mathbb{N} \setminus \{0\}\}$ sei endlich.

Sei q_m die grösste Primzahl in Q. Dann können wir jede Zahl n_i eindeutig als

$$n_i = q_1^{r_{i,1}} \cdot q_2^{r_{i,2}} \cdot \dots \cdot q_m^{r_{i,m}}$$

für irgendwelche $r_{i,1}, r_{i,2}, ..., r_{i,m} \in \mathbb{N}$ darstellen. Sei c die binäre Länge eines Programms, dass diese $r_{i,j}$ als Eingaben nimmt und n_i erzeugt (A ist für alle $i \in \mathbb{N}$ bis auf die Eingaben $r_{i,1}, ..., r_{i,m}$ gleich).

Dann gilt:

$$K(n_i) \leq c + 8 \cdot (\lceil \log_2(r_{i,1} + 1) \rceil + \lceil \log_2(r_{i,2} + 1) \rceil + ... + \lceil \log_2(r_{i,m} + 1) \rceil)$$

Die multiplikative Konstante 8 kommt daher, dass wir für die Zahlen $r_{i,1}, r_{i,2}, ..., r_{i,m}$ dieselbe Kodierung, wie für den Rest des Programmes verwenden (z.B. ASCII-Kodierung), damit ihre Darstellungen eindeutig voneinander getrennt werden können. Weil $r_{i,j} \leq \log_2 n_i, \forall j \in \{1, ..., m\}$ erhalten wir

$$K(n_i) \leq c + 8m \cdot \lceil \log_2(\log_2 n_i + 1) \rceil, \forall i \in \mathbb{N} \setminus \{0\}$$

Weil m und c Konstanten unabhängig von i sind, kann

$$\lceil \log_2 n_i \rceil / 2 \leq K(n_i) \leq c + 8m \cdot \lceil \log_2(\log_2 n_i + 1) \rceil$$

$$\lceil \log_2 n_i \rceil / 2 \leq c + 8m \cdot \lceil \log_2(\log_2 n_i + 1) \rceil$$

nur für endlich viele $i \in \mathbb{N} \setminus \{0\}$ gelten.

Dies ist ein Widerspruch!

Folglich ist die Menge Q unendlich.

■

3.2 How To Kolmogorov

Aufgabentyp 1

Sei $w_n = (010)^{3^{2n^3}} \in \{0,1\}^*$ für alle $n \in \mathbb{N} \setminus \{0\}$. Gib eine möglichst gute obere Schranke für die Kolmogorov-Komplexität von w_n an, gemessen in der Länge von w_n.

Lösung Typ 1

Wir zeigen ein Programm, dass n als Eingabe nimmt und w_n druckt:

```
W_n:    begin
        M := n;
        M := 2 × M × M × M;
        J := 1;
        for I = 1 to M
            J := J × 3;
        for I = 1 to J;
            write(010);
    end
```

Der einzige variable Teil dieses Algorithmus ist n. Der restliche Code ist von konstanter Länge. Die binäre Länge dieses Programms kann von oben durch

$$\lceil \log_2(n + 1) \rceil + c$$

beschränkt werden, für eine Konstante c.

Somit folgt

$$K(w_n) \leq \log_2(n) + c'$$

Wir berechnen die Länge von w_n als $|w_n| = |010| \cdot 3^{2n^3} = 3^{2n^3+1}$.

Mit ein wenig umrechnen erhalten wir

$$n = \sqrt[3]{\frac{\log_3 |w_n| - 1}{2}}$$

und die obere Schranke

$$K(w_n) \leq \log_2 \left(\sqrt[3]{\frac{\log_3 |w_n| - 1}{2}} \right) + c' \leq \log_2 \log_3 |w_n| + c''$$

Aufgabentyp 2

Geben Sie eine unendliche Folge von Wörtern $y_1 < y_2 < ...$ an, so dass eine Konstante $c \in \mathbb{N}$ existiert, so dass für alle $i \geq 1$

$$K(y_i) \leq \log_2 \log_2 \log_3 \log_2(|y_i|) + c$$

Lösung Typ 2

Wir definieren die Folge $y_1, y_2, ...$ mit $y_i = 0^{2^{3^{2^i}}}$ für alle $i \in \mathbb{N}$. Da $|y_i| < |y_{i+1}|$ folgt die geforderte Ordnung.

Es gilt

$$i = \log_2 \log_3 \log_2 |y_i| \text{ für } i \geq 1$$

Wir zeigen ein Programm, dass i als Eingabe nimmt und y_i druckt:

```
begin
    M := i;
    M := 2^(3^(2^M));
    for I = 1 to M;
        write(010);
end
```

Das ^ für die Exponentiation ist nicht Teil der originalen Pascal Syntax, aber wir verwenden es um unser Programm lesbarer zu machen.

Der einzige variable Teil dieses Programms ist das i. Der Rest hat konstante Länge. Demnach kann die Länge diese Programms für eine Konstante c' durch

$$\lceil \log_2(i+1) \rceil + c'$$

von oben beschränkt werden.

Somit folgt

$$\begin{aligned} K(y_i) &\leq \log_2(i) + c \\ &\leq \log_2 \log_2 \log_3 \log_2 |y_i| + c \end{aligned}$$

für eine Konstante c.

Aufgabentyp 3

Sei $M = \{7^i \mid i \in \mathbb{N},\ i \leq 2^n - 1\}$. Beweisen Sie, dass mindestens sieben Achtel der Zahlen in M Kolmogorov-Komplexität von mindestens $n - 3$ haben.

Lösung Typ 3

Wir zeigen, dass höchstens $\frac{1}{8}$ der Zahlen $x \in M$ eine Kolmogorov-Komplexität $K(x) \leq n-4$ haben.

Nehmen wir zum Widerspruch an, dass M mehr als $\frac{1}{8}|M|$ Zahlen x enthält, mit $K(x) \leq n-4$.

Die Programme, die diese Wörter generieren, müssen paarweise verschieden sein, da die Wörter paarweise verschieden sind.

Es gibt aber höchstens

$$\sum_{k=0}^{n-4} 2^k = 2^{n-3} - 1 < \frac{1}{8}|M|$$

Bitstrings mit Länge $\leq n-4$. **Widerspruch.**

4 Endliche Automaten - Einführung

4.1 Erster Ansatz zur Modellierung von Algorithmen

Ein (deterministischer) **endlicher Automat (EA)** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$, wobei

(i) Q eine endliche Menge von **Zuständen** ist,

(ii) Σ ein Alphabet, genannt **Eingabealphabet**, ist,

(iii) $q_0 \in Q$ der Anfangszustand ist,

(iv) $F \subseteq Q$ die **Menge der akzeptierenden Zustände** ist und

(v) $\delta : Q \times \Sigma \to Q$ die **Übergangsfunktion** ist.

Konfigurationen

Eine **Konfiguration** von M ist ein Tupel $(q, w) \in Q \times \Sigma^*$.

- "M befindet sich in einer Konfiguration $(q, w) \in Q \times \Sigma^*$, wenn M im Zustand q ist und noch das Suffix w eines Eingabewortes lesen soll."
- Die Konfiguration $(q_0, x) \in \{q_0\} \times \Sigma^*$ heisst die **Startkonfiguration von M auf x**.
- Jede Konfiguration aus $Q \times \{\lambda\}$ nennt man **Endkonfiguration**.

Ein **Schritt** von M ist eine Relation (auf Konfigurationen) $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$, definiert durch

$$(q, w) \vdash_M (p, x) \iff w = ax, a \in \Sigma \text{ und } \delta(q, a) = p.$$

Berechnungen

Eine **Berechnung** C von M ist eine endliche Folge $C = C_0, C_1, ..., C_n$ von Konfigurationen, so dass

$$C_i \vdash_M C_{i+1} \text{ für alle } 0 \leq i \leq n-1.$$

C ist die **Berechnung von M auf einer Eingabe** $x \in \Sigma^*$, falls $C_0 = (q_0, x)$ und $C_n \in Q \times \{\lambda\}$ eine Endkonfiguration ist.

Falls $C_n \in F \times \{\lambda\}$, sagen wir, dass C eine **akzeptierende Berechnung** von M auf x ist, und dass M **das Wort x akzeptiert**.

Falls $C_n \in (Q \setminus F) \times \{\lambda\}$, sagen wir, dass C eine **verwerfende Berechnung** von M auf x ist, und dass M **das Wort** x **verwirft (nicht akzeptiert)**.

Transitivität von $\vdash_M$ und δ

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat. Wir definieren $\vdash_M^*$ als die reflexive und transitive Hülle der Schrittrelation $\vdash_M$ von M; daher ist

$$(q, w) \vdash_M^* (p, u) \iff (q = p \wedge w = u) \text{ oder } \exists k \in \mathbb{N} \setminus \{0\},$$

so dass

(i) $w = a_1 a_2 ... a_k u, a_i \in \Sigma$ für $i = 1, 2, ..., k$, und

(ii) $\exists r_1, r_2, ..., r_{k-1} \in Q$, so dass

$$(q, w) \vdash_M (r_1, a_2 ... a_k u) \vdash_M ... \vdash_M (r_{k-1}, a_k u) \vdash_M (p, u)$$

Wir definieren $\hat{\delta} : Q \times \Sigma^* \to Q$ durch:

(i) $\hat{\delta}(q, \lambda) = q$ für alle $q \in Q$ und

(ii) $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ für alle $a \in \Sigma, w \in \Sigma^*, q \in Q$.

$$\hat{\delta}(q, w) = p \iff (q, w) \vdash_M^* (p, \lambda)$$

4.2 Reguläre Sprachen

Die **von** M **akzeptierte Sprache** $L(M)$ ist definiert als

$$\begin{aligned} L(M) &= \{w \in \Sigma^* \mid \text{Berechnung von } M \text{ auf } w \text{ endet in } (p, \lambda) \in F \times \{\lambda\}\} \\ &= \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda) \wedge p \in F\} \\ &= \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\} \end{aligned}$$

$\mathcal{L}_{\mathbf{EA}} = \{L(M) \mid M \text{ ist ein EA}\}$ ist die Klasse der Sprachen, die von endlichen Automaten akzeptiert werden.
$\mathcal{L}_{\text{EA}}$ bezeichnet man auch als die **Klasse der regulären Sprachen**, und jede Sprache $L \in \mathcal{L}_{\text{EA}}$ wird **regulär** genannt.

Klassen für alle Zustände im Endlichen Automaten

Für alle $p \in Q$ definieren wir die Klasse

$$\begin{aligned} \mathbf{Kl[p]} &= \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) = p\} \\ &= \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda)\} \end{aligned}$$

Wir bemerken dann

$$\bigcup_{q \in Q} \text{Kl[q]} = \Sigma^*$$

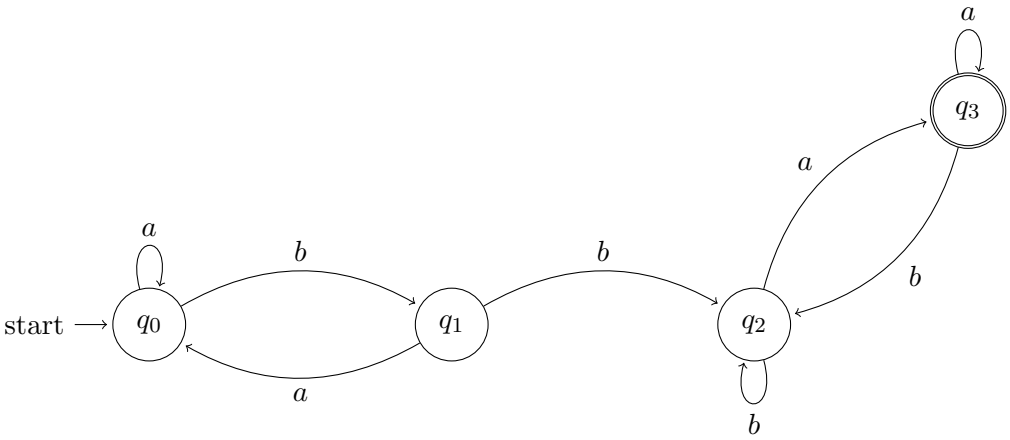
$$\text{Kl}[q] \cap \text{Kl}[p] = \emptyset, \forall p, q \in Q, p \neq q$$

$$L(M) = \bigcup_{q \in F} \text{Kl}[q]$$

EA Konstruktion - Beispielaufgabe

Entwerfen sie für folgende Sprache einen Endlichen Automat und geben Sie eine Beschreibung von Kl[q] für jeden Zustand $q \in Q$.

$$L_1 = \{xbbya \in \{a,b\}^* \mid x, y \in \{a,b\}^*\}$$



Wir beschreiben nun die Klassen für die Zustände q_0, q_1, q_2, q_3:
Kl[q_0] = $\{wa \in \{a,b\}^* \mid$ Das Wort w enthält nicht die Teilfolge $bb\} \cup \{\lambda\}$
Kl[q_1] = $\{wb \in \{a,b\}^* \mid$ Das Wort w enthält nicht die Teilfolge $bb\}$
Kl[q_3] = $\{wa \in \{a,b\}^* \mid$ Das Wort w enthält die Teilfolge $bb\} = L_1$
Kl[q_2] = $\{a,b\}^* - (\text{Kl}[q_0] \cup \text{Kl}[q_1] \cup \text{Kl}[q_3])$

4.3 Produktautomaten - Simulationen

Lemma 3.2

Sei Σ ein Alphabet und seien $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ und $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ zwei EA. Für jede Mengenoperation $\odot \in \{\cup, \cap, -\}$ existiert ein EA M, so dass

$$L(M) = L(M_1) \odot L(M_2).$$

Sei $M = (Q, \Sigma, \delta, q_0, F_\odot)$, wobei

(i) $Q = Q_1 \times Q_2$

(ii) $q_0 = (q_{01}, q_{02})$

(iii) für alle $q \in Q_1$, $p \in Q_2$ und $a \in \Sigma$, $\delta((q,p),a) = (\delta_1(q,a), \delta_2(p,a))$,

(iv) falls $\odot = \cup$, dann ist $F = F_1 \times Q_2 \cup Q_1 \times F_2$
falls $\odot = \cap$, dann ist $F = F_1 \times F_2$, und
falls $\odot = -$, dann ist $F = F_1 \times (Q_2 - F_2)$.

Produktautomat - Beispielaufgabe

Verwenden Sie die Methode des modularen Entwurfs (Konstruktion eines Produktautomaten), um einen endlichen Automaten (in Diagrammdarstellung) für die Sprache

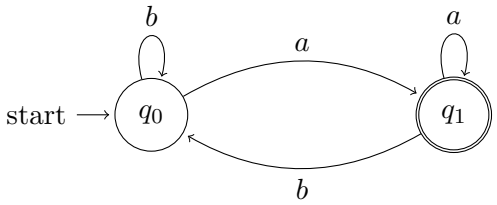
$$L = \{w \in \{a,b\}^* \mid |w|_a = 2 \text{ oder } w = ya\}$$

zu entwerfen. Zeichnen Sie auch jeden der Teilautomaten und geben Sie für die Teilautomaten für jeden Zustand q die Klasse Kl[q] an.

Wir teilen L wie folgt auf:

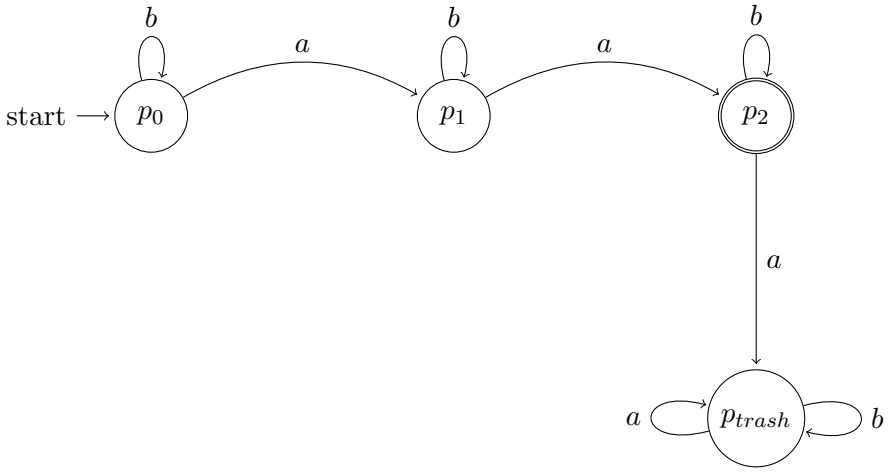
$$L = L_1 \cup L_2 \text{ wobei gilt:}$$
$$L_1 = \{w \in \{a,b\}^* \mid w = ya\}$$
$$L_2 = \{w \in \{a,b\}^* \mid |w|_a = 2\}$$

Zuerst zeichnen wir die 2 einzelnen Teilautomaten und geben für jeden Zustand q bzw. p die Klasse Kl[q] respektive Kl[p] an:



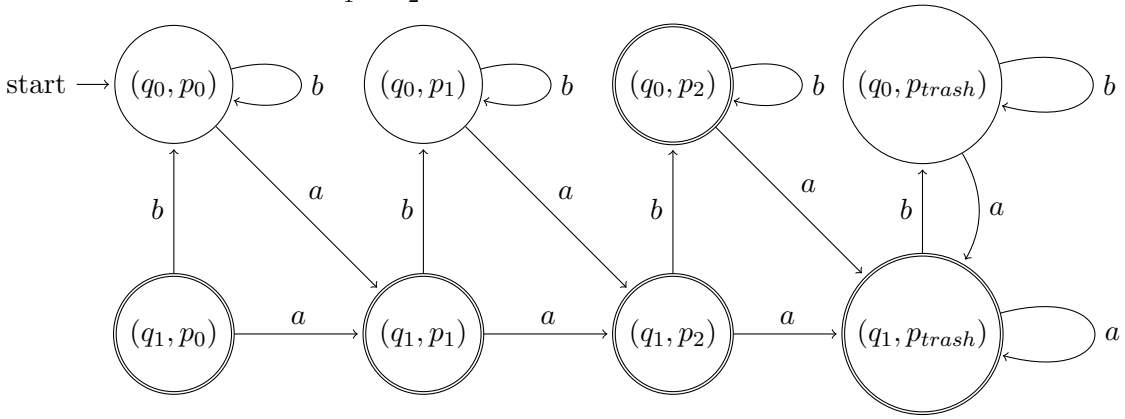
erster Teilautomat: $L_1 = \{w \in \{a,b\}^* \mid w = ya\}$
Wir beschreiben nun die Zustände für die Klassen q_0 und q_1:
Kl[q_0] = $\{yb \mid y \in \{a,b\}^*\} \cup \{\lambda\}$
Kl[q_1] = $\{ya \mid y \in \{a,b\}^*\}$

zweiter Teilautomat: $L_2 = \{w \in \{a,b\}^* \mid |w|_a = 2\}$



Wir beschreiben nun die Zustände für die Klassen p_0, p_1, p_2, p_{trash}: Kl[p_0] = $\{w \in \{a,b\}^* \mid |w|_a = 0\}$
Kl[p_1] = $\{w \in \{a,b\}^* \mid |w|_a = 1\}$
Kl[p_2] = $\{w \in \{a,b\}^* \mid |w|_a = 2\}$
Kl[p_{trash}] = $\{w \in \{a,b\}^* \mid |w|_a > 2\}$

Zum Schluss kombinieren wir diese Teilautomaten zu einem Produktautomaten:
Produktautomat: $L = L_1 \cup L_2$



5 Beweise für Nichtregularität

5.1 Einführung und grundlegende Tipps

i. Wichtiges Unterkapitel. Kommt fast garantiert am Midterm.

ii. Um $L \notin \mathcal{L}_{\text{EA}}$ zu zeigen, genügt es zu beweisen, dass es keinen EA gibt, der L akzeptiert.

iii. Nichtexistenz ist generell sehr schwer zu beweisen, da aber die Klasse der endlichen Automaten sehr eingeschränkt ist, ist dies nicht so schwierig.

iv. Wir führen Widerspruchsbeweise.

v. Es gibt 3 Arten Nichtregularitätsbeweise zu führen (Lemma 3.3, Pumping-Lemma und Kolmogorov-Komplexität).

vi. Ihr müsst alle 3 Methoden können. Ist aber halb so wild.

5.2 Theorie für Nichtregularitätsbeweise

5.2.1 Lemma 3.3 Methode

Lemma 3.3

Sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein EA. Seien $x, y \in \Sigma^*, x \neq y$, so dass

$$\hat{\delta}_A(q_0, x) = p = \hat{\delta}_A(q_0, y)$$

für ein $p \in Q$ (also $x, y \in \text{Kl}[p]$). Dann existiert für jedes $z \in \Sigma^*$ ein $r \in Q$, so dass xz und $yz \in \text{Kl}[r]$, also gilt insbesondere

$$xz \in L(A) \iff yz \in L(A)$$

Beweis:

Aus der Existenz der Berechnungen

$(q_0, x) \vdash_A^* (p, \lambda)$ und $(q_0, y) \vdash_A^* (p, \lambda)$ von A folgt die Existenz der Berechnungen auf xz und yz:

$(q_0, xz) \vdash_A^* (p, z)$ und $(q_0, yz) \vdash_A^* (p, z)$ für alle $z \in \Sigma^*$.

Wenn $r = \hat{\delta}_A(p, z)$ ist, dann ist die Berechnung von A auf xz und yz:

$(q_0, xz) \vdash_A^* (p, z) \vdash_A^* (r, \lambda)$ und $(q_0, yz) \vdash_A^* (p, z) \vdash_A^* (r, \lambda)$.

Wenn $r \in F$, dann sind beide Wörter xz und yz in $L(A)$. Falls $r \notin F$, dann sind $xz, yz \notin L(A)$.

■

Bemerkungen

- Von den 3 vorgestellten Methoden, ist diese Methode die einzige, die (unter der richtigen Anwendung) garantiert für jede nichtreguläre Sprache funktioniert.
- Um die Nichtregularität von L zu beweisen, verwenden wir die Endlichkeit von Q und das Pigeonhole-Principle.

Beispielaufgabe - Lemma 3.3

Betrachten wir mal eine Beispielaufgabe mit dieser Methode am Paradebeispiel

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

Nehmen wir zum Widerspruch an L sei regulär.

Dann existiert ein EA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$.

Wir betrachten die Wörter $0^1, \ldots, 0^{|Q|+1}$. Per Pigeonhole-Principle existiert O.B.d.A. $i < j$, so dass

$$\hat{\delta}(q_0, 0^i) = \hat{\delta}(q_0, 0^j)$$

Nach Lemma 3.3 gilt

$$0^i z \in L \iff 0^j z \in L$$

für alle $z \in (\Sigma_{\text{bool}})^*$. Dies führt aber zu einem Widerspruch, weil für $z = 1^i$ das Wort $0^i 1^i \in L$ aber $0^j 1^i \notin L$.

5.2.2 Pumping Lemma Methode

Pumping Lemma

Sei L regulär. Dann existiert eine Konstante $n_0 \in \mathbb{N}$, so dass jedes Wort $w \in \Sigma^*$ mit $|w| \geq n_0$ in drei Teile x, y und z zerlegen lässt, das heisst $w = yxz$, wobei

(i) $|yx| \leq n_0$

(ii) $|x| \geq 1$

(iii) entweder $\{yx^k z \mid k \in \mathbb{N}\} \subseteq L$ oder $\{yx^k z \mid k \in \mathbb{N}\} \cap L = \emptyset$.

Beweis

Sei $L \in \Sigma^*$ regulär. Dann existiert ein EA $A = (Q, \Sigma, \delta_A, q_0, F)$, so dass $L(A) = L$.

Sei $n_0 = |Q|$ und $w \in \Sigma^*$ mit $|w| \geq n_0$. Dann ist $w = w_1 w_2 ... w_{n_0} u$, wobei $w_i \in \Sigma$ für $i = 1, ..., n_0$ und $u \in \Sigma^*$. Betrachten wir die Berechnung auf $w_1 w_2 ... w_{n_0}$:

$$(q_0, w_1 w_2 w_3 ... w_{n_0}) \vdash_A (q_1, w_2 w_3 ... w_{n_0}) \vdash_A ... \vdash_A (q_{n_0-1}, w_{n_0}) \vdash_A (q_{n_0}, \lambda)$$

In dieser Berechnung kommen n_0+1 Zustände $q_0, q_1, ..., q_{n_0}$ vor. Da $|Q| = n_0$, existieren $i, j \in \{0, 1, ..., n_0\}, i < j$, so dass $q_i = q_j$. Daher haben wir in der Berechnung die Konfigurationen

$$(q_0, w_1 w_2 w_3 ... w_{n_0}) \vdash_A^* (q_i, w_{i+1} w_{i+2} ... w_{n_0}) \vdash_A^* (q_i, w_{j+1} ... w_{n_0}) \vdash_A^* (q_{n_0}, \lambda)$$

Dies impliziert

$$(q_i, w_{i+1} w_{i+2} ... w_j) \vdash_A^* (q_i, \lambda) \qquad (1)$$

Wir setzen nun $y = w_1 ... w_i$, $x = w_{i+1} ... w_j$ und $z = w_{j+1} ... w_{n_0} u$, so dass $w = yxz$.

Wir überprüfen nun die Eigenschaften (i),(ii) und (iii):

(i) $yx = w_1 ... w_i w_{i+1} ... w_j$ und daher $|yx| = j \leq n_0$.

(ii) Da $|x| \geq j - i$ und $i < j$, ist $|x| \geq 1$.

(iii) (1) impliziert $(q_i, x^k) \vdash_A^* (q_i, \lambda)$ für alle $k \in \mathbb{N}$. Folglich gilt für alle $k \in \mathbb{N}$:

$$(q_0, yx^k z) \vdash_A^* (q_i, x^k z) \vdash_A^* (q_i, z) \vdash_A^* (\hat{\delta}_A(q_i, z), \lambda)$$

Wir sehen, dass für alle $k \in \mathbb{N}$ die Berechnungen im gleichen Zustand $q_{end} = \hat{\delta}_A(q_i, z)$ enden. Falls also $q_{end} \in F$, akzeptiert A alle Wörter aus $\{yx^k z \mid k \in \mathbb{N}\}$. Falls $q_{end} \notin F$, dann akzeptiert A kein Wort aus $\{yx^k z \mid k \in \mathbb{N}\}$.

■

Beispielaufgabe - Pumping Lemma

Versuchen wir zu beweisen, dass

$$L_2 = \{wabw^{\mathbf{R}} \mid w \in \{a,b\}^*\}$$

nicht regulär ist.

Wir nehmen zum Widerspruch an, dass L_2 regulär ist.

Das Pumping-Lemma (Lemma 3.4) besagt, dass dann eine Konstante $n_0 \in \mathbb{N}$ existiert, so dass sich jedes Wort $w \in \Sigma*$ mit $|w| \geq n_0$ in drei Teile y, x, und z zerlegen lässt. ($\implies w = yxz$). Wobei folgendes gelten muss:

(i) $|yx| \leq n_0$

(ii) $|x| \geq 1$

(iii) **entweder** $\{yx^kz \mid k \in \mathbb{N}\} \subseteq L_2$ **oder** $\{yx^kz \mid k \in \mathbb{N}\} \cap L_2 = \emptyset$

Wir wählen $w = a^{n_0}aba^{n_0}$. Es ist leicht zu sehen das $|w| = 2n_0 + 2 \geq n_0$.

Da nach (i), $|yx| \leq n_0$ gelten muss, haben wir $y = a^l$ und $x = a^m$ für beliebige $l, m \in \mathbb{N}, l + m \leq n_0$.
Somit gilt $z = a^{n_0-(l+m)}aba^{n_0}$

Nach (ii) ist $m \geq 1$.

Wir haben also $\{yx^kz \mid k \in \mathbb{N}\} = \{a^{n_0-m+km}aba^{n_0} \mid k \in \mathbb{N}\}$

Da $yx^1z = a^{n_0}aba^{n_0}$ und

$a^{n_0}aba^{n_0} \in \{a^{n_0-m+km}aba^{n_0} \mid k \in \mathbb{N}\} \wedge a^{n_0}aba^{n_0} \in L_2$ gilt, folgt

$$\{a^{n_0-m+km}aba^{n_0} \mid k \in \mathbb{N}\} \cap L_2 \neq \emptyset$$

Wenn wir nun $k = 0$ wählen und uns daran erinnern, dass $m \geq 1$, erhalten wir folgendes

$$\Rightarrow yx^0z = yz = a^{n_0-m}aba^{n_0} \notin L_2$$

Daraus folgt,

$$\{a^{n_0-m+km}aba^{n_0} \mid k \in \mathbb{N}\} \not\subseteq L_2$$

Somit gilt (iii) nicht.

Dies ist ein Widerspruch! Somit haben wir gezeigt, dass die Sprache $L_2 = \{wabw^{\mathbf{R}} \mid w \in \{a,b\}^*\}$ nicht regulär ist.

5.2.3 Kolmogorov Methode

Satz 3.1

Sei $L \subseteq (\Sigma_{\text{bool}})^*$ eine reguläre Sprache. Sei $L_x = \{y \in (\Sigma_{\text{bool}})^* \mid xy \in L\}$ für jedes $x \in (\Sigma_{\text{bool}})^*$. Dann existiert eine Konstante **const**, so dass für alle $x, y \in (\Sigma_{\text{bool}})^*$

$$K(y) \leq \lceil \log_2(n+1) \rceil + \textbf{ const},$$

falls y das n-te Wort in der Sprache L_x ist.

Wie wir sehen werden, beruht der Nichtregularitätsbeweis darauf, dass die Differenz von $|w_{n+1}| - |w_n|$ für kanonische Wörter $(w_i)_{i\in\mathbb{N}}$ beliebig gross werden kann.

Beispielaufgabe - Kolmogorov Methode

Verwenden Sie die Methode der Kolmogorov-Komplexität, um zu zeigen, dass die Sprache

$$L_1 = \{0^{n^2 \cdot 2^n} \mid n \in \mathbb{N}\}$$

nicht regulär ist.

Angenommen L_1 sei regulär.

Wir betrachten

$$L_{0^{m^2 \cdot 2^m + 1}} = \{y \mid 0^{m^2 \cdot 2^m + 1} y \in L_1\}.$$

Da

$$\begin{aligned}(m+1)^2 \cdot 2^{m+1} &= (m^2 + 2m + 1) \cdot 2^{m+1} \\ &= m^2 \cdot 2^m + m^2 \cdot 2^m + (2m+1) \cdot 2^{m+1} \\ &= m^2 \cdot 2^m + (m^2 + 4m + 2) \cdot 2^m\end{aligned}$$

ist für jedes $m \in \mathbb{N}$ das Wort $y_1 = 0^{(m^2+4m+2) \cdot 2^m - 1}$ das kanonisch erste Wort der Sprache $L_{0^{m^2 \cdot 2^m + 1}}$.

Nach Satz 3.1 existiert eine Konstante c, unabhängig von m, so dass

$$K(y_1) \leq \lceil \log_2(1+1) \rceil + c = 1 + c.$$

Die Anzahl aller Programme, deren Länge kleiner oder gleich $1 + c$ sind, ist endlich.

Da es aber unendlich viel Wörter der Form $0^{(m^2+4m+2) \cdot 2^m - 1}$ gibt, ist dies ein Widerspruch.

Demzufolge ist L_1 nicht regulär.

■

5.3 Weitere Aufgaben

Beispielaufgabe 1 - Direkte Methode (Lemma 3.3)

Verwende eine direkte Argumentation über den Automaten (unter Verwendung von Lemma 3.3), um zu zeigen, dass die Sprache

$$L_2 = \{w \in \{0,1\}^* \mid |u|_0 \leq |u|_1 \text{ für alle Präfixe } u \text{ von } w\}$$

nicht regulär ist.

Angenommen L_2 sei regulär.

Dann existiert ein Endlicher Automat $A = (Q, \{0,1\}, \delta, q_0, F)$ mit $L(A) = L_2$.

Wir betrachten die Wörter

$$1, 1^2, ..., 1^{|Q|+1}$$

Per Pigeonhole-Principle existiert $i, j \in \{1, ..., |Q| + 1\}$ mit $i < j$, so dass

$$\hat{\delta}(q_0, 1^i) = \hat{\delta}(q_0, 1^j).$$

Nach Lemma 3.3 gilt nun für alle $z \in \{0,1\}^*$

$$1^i z \in L_2 \iff 1^j z \in L_2$$

Sei $z = 0^j$. Wir haben dann also

$$1^i z = 1^i 0^j \notin L_2,$$

da $i < j$ und ein Wort auch ein Präfix von sich selbst ist (Die Bedingung $|1^i 0^j|_0 \leq |1^i 0^j|_1$ wird verletzt). Aber wir haben auch

$$1^j z = 1^j 0^j \in L_2,$$

was zu einem Widerspruch führt. Also ist die Annahme falsch und L_2 nicht regulär.

■

Einschub - Sprachen mit Einsymbolalphabet

Angenommen es handelt sich bei $L \subseteq \Sigma^*$ um eine Sprache über einem unären Alphabet ($|\Sigma| = 1, \Sigma = \{x\}$).
Dann gilt:

$$\forall w \in \Sigma^* : w = x^{|w|}$$

Insbesondere gibt es für jede Länge nur ein Wort.
Sei die Folge $(w_i)_{i\in\mathbb{N}}$ kanonisch geordnet, so dass $w_i \in L$ (Wenn L endlich betrachten wir nur endlich viele Wörter der Folge).
Durch das gilt folgendes

$$\forall w \in \Sigma^*.\ \forall k \in \mathbb{N}.\ |w_k| < |w| < |w_{k+1}| \implies w \notin L$$

Beispielaufgabe 2 - Pumping Lemma

Zeigen Sie, dass

$$L = \{0^{n\cdot\lceil\sqrt{n}\rceil} \mid n \in \mathbb{N}\}$$

nicht regulär ist.

Angenommen $L = \{0^{0\cdot\lceil\sqrt{0}\rceil}, 0^{1\cdot\lceil\sqrt{1}\rceil}, 0^{2\cdot\lceil\sqrt{2}\rceil}, ...\}$ sei regulär.

Seien $w_0, w_1, w_2, ...$ die Wörter von L in kanonischer Reihenfolge. Nach dem Pumping Lemma gibt es ein $n_0 \in \mathbb{N}$, dass die Bedingungen (i)-(iii) erfüllt sind.

Wir wählen $w = w_{n_0^2} = 0^{n_0^2\lceil\sqrt{n_0^2}\rceil} \in L$.

Es ist leicht zu sehen das $|w| \geq n_0$ und folglich existiert eine Aufteilung $w = yxz$ ($y = 0^l$, $x = 0^m$ und $z = 0^{n_0^2\lceil\sqrt{n_0^2}\rceil - l - m}$), die (i)-(iii) erfüllt.

Da nach (i) $|yx| = l + m \leq n_0$, folgt $|x| = m \leq n_0$.

Aus (ii) folgt $|x| = m \geq 1$.

Wegen $|yx^2z| = |yxz| + |x|$ gilt also $|yxz| < |yx^2z| \leq |yxz| + n_0$.

Das nächste Wort in L nach $w_{n_0^2}$ ist $w_{n_0^2+1}$ und es gilt

$$\begin{aligned}
|w_{n_0^2+1}| - |w_{n_0^2}| &= (n_0^2+1)\cdot\lceil\sqrt{n_0^2+1}\rceil - n_0^2\cdot\lceil\sqrt{n_0^2}\rceil \\
&= (n_0^2+1)\cdot\lceil\sqrt{n_0^2+1}\rceil - n_0^2\cdot n_0 \\
&> (n_0^2+1)\cdot n_0 - n_0^3 \\
&= n_0
\end{aligned}$$

Die strikte Ungleichung gilt da $n_0 \in \mathbb{N}$ und $n_0 = \left\lceil\sqrt{n_0^2}\right\rceil < \sqrt{n_0^2+1} \leq \left\lceil\sqrt{n_0^2+1}\right\rceil$.

$$\implies |w_{n_0^2+1}| \geq |w_{n_0^2}| + (n_0+1)$$

Somit gilt

$$|w_{n_0^2}| < |yx^2z| < |w_{n_0^2+1}|$$

Daraus folgt $yx^2z \notin L$, während $yxz \in L$, in Widerspruch zu (iii).

■

Beispielaufgabe 3 - Kolmogorov Methode

Zeigen Sie, dass

$$L = \{0^{n \cdot \lceil \sqrt{n} \rceil} \mid n \in \mathbb{N}\}$$

nicht regulär ist.

Widerspruchsannahme: Sei L regulär.

Wir betrachten

$$L_{0^{m \cdot \lceil \sqrt{m} \rceil + 1}} = \{y \in \Sigma^* \mid 0^{m \cdot \lceil \sqrt{m} \rceil + 1} y \in L\}$$

Dann ist für jedes $m \in \mathbb{N}$ das Wort

$$y_1 = 0^{(m+1) \cdot \lceil \sqrt{m+1} \rceil - (m \cdot \lceil \sqrt{m} \rceil + 1)}$$

das kanonisch erste Wort der Sprache $L_{0^{m \cdot \lceil \sqrt{m} \rceil + 1}}$.

Nach Satz 3.1 existiert eine Konstante c, so dass gilt

$$K(y_1) \leq \lceil \log_2(1+1) \rceil + c = 1 + c$$

für jedes $m \in \mathbb{N}$.

Da die Länge von $|y_1|$

$$\begin{aligned} |y_1| &= (m+1) \cdot \lceil \sqrt{m+1} \rceil - (m \cdot \lceil \sqrt{m} \rceil + 1) \\ &\geq (m+1) \cdot \lceil \sqrt{m} \rceil - m \cdot \lceil \sqrt{m} \rceil - 1 \\ &= \lceil \sqrt{m} \rceil - 1 \overset{m \to \infty}{\longrightarrow} \infty \end{aligned}$$

beliebig gross werden kann, gibt es unendlich viele Wörter von dieser Form.

Dies ist ein Widerspruch, da es nur endlich viele Programme der Länge maximal $1 + c$ geben kann.

■

6 Nichtdeterministische Endliche Automaten

6.1 Definitionen

Definition NEA

Ein **nichtdeterministischer endlicher Automat (NEA)** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$. Dabei ist

(i) Q eine endliche Menge, **Zustandsmenge** genannt,

(ii) Σ ein Alphabet, **Eingabealphabet** genannt,

(iii) $q_0 \in Q$ der **Anfangszustand**,

(iv) $F \subseteq Q$ die Menge der **akzeptierenden Zustände** und

(v) δ eine Funktion von $Q \times \Sigma$ nach $\mathcal{P}(Q)$, **Übergangsfunktion genannt**.

Ein NEA kann zu einem Zustand q und einem gelesenen Zeichen a mehrere oder gar keinen Nachfolgezustand haben.

Konfigurationen für NEAs

Eine **Konfiguration** von M ist ein Tupel $(q, w) \in Q \times \Sigma^*$.

- "M befindet sich in einer Konfiguration $(q, w) \in Q \times \Sigma^*$, wenn M im Zustand q ist und noch das Suffix w eines Eingabewortes lesen soll."

- Die Konfiguration $(q_0, x) \in \{q_0\} \times \Sigma^*$ ist die **Startkonfiguration für das Wort** x.

Ein **Schritt** von M ist eine Relation (auf Konfigurationen) $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$, definiert durch

$$(q, w) \vdash_M (p, x) \iff w = ax, a \in \Sigma \text{ und } p \in \delta(q, a)$$

Berechnungen für NEAs

Eine **Berechnung von M** ist eine endliche Folge $C_1, ..., C_k$ von Konfigurationen, so dass

$$C_i \vdash_M C_{i+1} \text{ für alle } 1 \leq i \leq k.$$

Eine **Berechnung von M auf x** ist eine Berechnung $C = C_0, ..., C_m$, wobei $C_0 = (q_0, x)$ und **entweder** $C_m \in Q \times \{\lambda\}$ **oder** $C_m = (q, ay)$ für ein $a \in \Sigma, y \in \Sigma^*$ und $q \in Q$, so dass $\delta(q, a) = \emptyset$.

Falls $C_m \in F \times \{\lambda\}$, sagen wir, dass C eine **akzeptierende Berechnung** von M auf x ist, und dass M **das Wort** x **akzeptiert**.

Die Relation $\vdash_M^*$ ist die reflexive und transitive Hülle von $\vdash_M$, genau wie bei einem EA.

Wir definieren

$$\mathbf{L(M)} = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda) \text{ für ein } p \in F\}$$

als die **von M akzeptierte Sprache.**

Zu der Übergangsfunktion δ definieren wir die Funktion $\hat{\delta} : (Q \times \Sigma^*) \to \mathcal{P}(Q)$ wie folgt:

(i) $\hat{\delta}(q, \lambda) = \{q\}$ für alle $q \in Q$

(ii) $\hat{\delta}(q, wa) = \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a)$ für alle $q \in Q, a \in \Sigma, w \in \Sigma^*$.

Repetition Pumping Lemma - Aufgabe mit Case Distinction (12.b)

Wir zeigen per Pumping Lemma, dass die Sprache

$$L = \{w \in \{a, b, c\}^* \mid w \text{ enthält das Teilwort } ab \text{ gleich oft wie das Teilwort } ba\}$$

nicht regulär ist.

Zur Erinnerung:

Pumping Lemma

Sei L regulär. Dann existiert eine Konstante $n_0 \in \mathbb{N}$, so dass jedes Wort $w \in \Sigma^*$ mit $|w| \geq n_0$ in drei Teile x, y und z zerlegen lässt, das heisst $w = yxz$, wobei

(i) $|yx| \leq n_0$

(ii) $|x| \geq 1$

(iii) entweder $\{yx^kz \mid k \in \mathbb{N}\} \subseteq L$ oder $\{yx^kz \mid k \in \mathbb{N}\} \cap L = \emptyset$.

Lösung

Sei L regulär.

Nach dem Pumping Lemma existiert eine Konstante $n_0 \in \mathbb{N}$, so dass jedes Wort w mit $|w| \geq n_0$ die Bedingung des PL erfüllt.

Sei $w = (abc)^{n_0}(bac)^{n_0}$. Offensichtlich gilt $|w| \geq n_0$. Nach dem PL existiert eine Zerlegung $w = yxz$, die (i), (ii) und (iii) erfüllt.

Da yxz die Bedingung (i) erfüllt, gilt $|yx| \leq n_0$. Insbesondere folgt daraus, dass x komplett in der ersten Hälfte (i.e. $(abc)^{n_0}$) enthalten ist.

Aus (ii) folgt weiter, dass x mindestens ein Buchstaben enthält.

Case Distinction

I. **Case $\mathbf{x = c}$**

In diesem Fall enthält $yx^0z = yz$ das Teilwort ba einmal mehr als ab.

Somit gilt in diesem Fall $yx^0z \notin L$.

II. **Case x enthält mindestens ein a oder b**

Wir betrachten $yx^0z = yz$. In diesem Fall bleibt die Anzahl der Teilwörter ba gleich oder erhöht sich. Da aber die Anzahl der Teilwörter ab um mindestens 1 kleiner wird, gilt $yx^0z \notin L$.

Da die Case Distinction alle Fälle abdeckt folgt für die Zerlegung $yx^0z \notin L$. Aus $yxz \in L$ ergibt sich somit ein Widerspruch.

Demnach ist die Annahme falsch und L nicht regulär.

■

NEA - Beispiel aus der Vorlesung

Wir betrachten folgenden NEA $M = (\{q_0, q_1, q_2\}, \Sigma_{\text{bool}}, \delta, q_0, \{q_2\})$

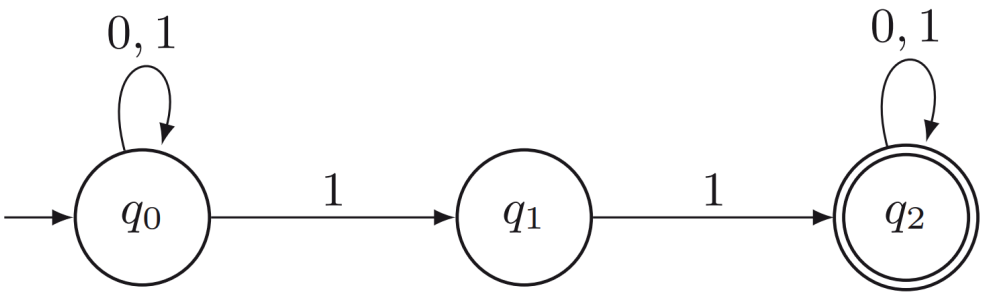


Figure 1: Abb. 3.15 aus dem Buch

Berechnungsbaum

Für ein Wort $x \in (\Sigma_{\text{bool}})^*$ ist ein Berechnungsbaum $\mathcal{B}_{\mathbf{M}}(\mathbf{x})$ nützlich, um zu erkennen, ob $x \in L(M)$.

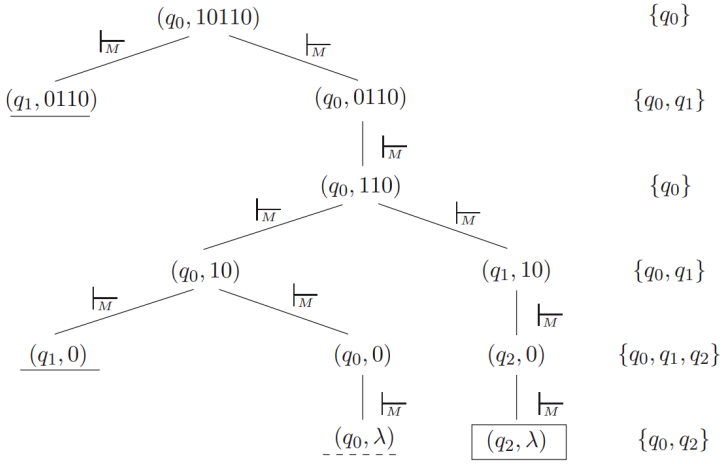


Figure 2: Abb. 3.16 aus dem Buch

Wir können die Sprache des NEA bestimmen.

Lemma 3.5

$$L(M) = \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}$$

Beweisidee

Beide Inklusionen zeigen und fertig. (Siehe Buch)

Wir definieren die Klasse $\mathcal{L}_{\text{NEA}}$.

$$\mathcal{L}_{\text{NEA}} = \{L(M) \mid M \text{ ist ein NEA}\}$$

6.2 Äquivalenz von NEA und EA

Beweis von $\mathcal{L}_{\text{NEA}} = \mathcal{L}_{\text{EA}}$ per **Potenzmengenkonstruktion**.

Satz 3.2

Zu jedem NEA M existiert ein EA A, so dass

$$L(M) = L(A)$$

Beweisidee

Potenzmengenkonstruktion und dann Induktion auf der Länge von einem Input i.e. $|x|$. (Siehe Buch)

Potenzmengenkonstruktion

Sei $M = (Q, \Sigma, \delta_M, q_0, F)$ ein NEA. Wir konstrurieren einen äquivalenten Endlichen Automaten $A = (Q_A, \Sigma_A, \delta_A, q_{0A}, F_A)$.

(i) $Q_A = \{\langle P\rangle \mid P \subseteq Q\}$

(ii) $\Sigma_A = \Sigma$

(iii) $q_{0A} = \langle\{q_0\}\rangle$

(iv) $F_A = \{\langle P\rangle \mid P \subseteq Q \text{ und } P \cap F \neq \emptyset\}$

(v) $\delta_A : (Q_A \times \Sigma_A) \to Q_A$ ist eine Funktion, definiert wie folgt. Für jedes $\langle P\rangle \in Q_A$ und jedes $a \in \Sigma_A$ ist

$$\begin{aligned}\delta_A(\langle P\rangle, a) &= \left\langle \bigcup_{p\in P} \delta_M(p,a) \right\rangle \\ &= \langle\{q \in Q \mid \exists p \in P, \text{ so dass } q \in \delta_M(p,a)\}\rangle\end{aligned}$$

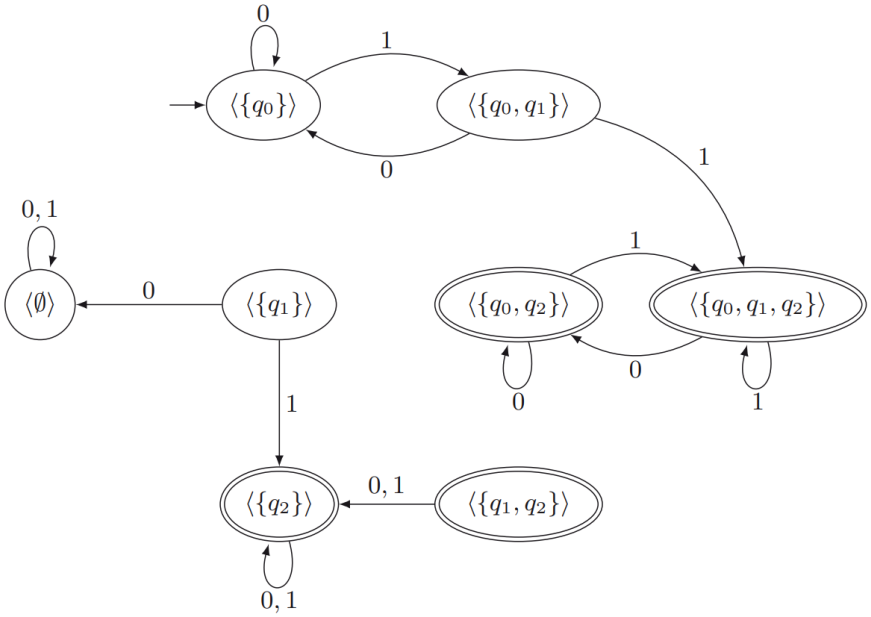


Figure 3: Abb. 3.18 im Buch

6.3 Exponentiell mehr Zustände - manchmal

Sei

$$L_k = \{x1y \mid x \in (\Sigma_{\text{bool}})^*,\ y \in (\Sigma_{\text{bool}})^{k-1}\}$$

Folgender NEA A_k mit $k+1$ akzeptiert L_k.

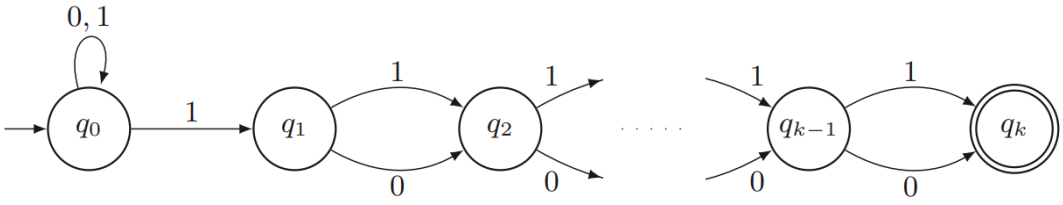


Figure 4: Abb. 3.19 im Buch

Lemma 3.6

Für alle $k \in \mathbb{N} \setminus \{0\}$ muss jeder EA, der L_k akzeptiert, mindestens 2^k Zustände haben.

Beweis

Sei $B_k = (Q_k, \Sigma_{bool}, \delta_k, q_{0k}, F_k)$ ein EA mit $L(B_k) = L_k$.

Nach **Lemma 3.3** gilt für $x, y \in (\Sigma_{bool})^*$:

Wenn $\hat{\delta}_k(q_{0k}, x) = \hat{\delta}_k(q_{0k}, y)$, dann gilt für alle $z \in (\Sigma_{bool})^*$:

$$xz \in L(B_k) \iff yz \in L(B_k)$$

Die Idee des Beweises ist es, eine Menge S_k von Wörtern zu finden, so dass für keine zwei unterschiedlichen Wörter $x, y \in S_k$ die Gleichung $\hat{\delta}_k(q_{0k}, x) = \hat{\delta}_k(q_{0k}, y)$ gelten darf. Dann müsste B_k mindestens $|S_k|$ viele Zustände haben.

Wir wählen $S_k = (\Sigma_{bool})^k$ und zeigen, dass $\hat{\delta}_k(q_{0k}, u)$ paarweise unterschiedliche Zustände für alle $u \in S_k$ sind.

Wir beweisen dies per Widerspruch.

Seien $x = x_1x_2...x_k$ und $y = y_1y_2...y_k$ für $x_i, y_i \in \Sigma_{bool}, i \in \{1, ..., k\}$ zwei unterschiedliche Wörter aus S_k.

Nehmen wir zum Widerspruch an, dass $\hat{\delta}_k(q_{0k}, x) = \hat{\delta}_k(q_{0k}, y)$.

Weil $x \neq y$, existiert ein $j \in \{1, ..., k\}$, so dass $x_j \neq y_j$. O.B.d.A. setzen wir $x_j = 1$ und $y_j = 0$. Betrachten wir nun $z = 0^{j-1}$. Dann ist

$xz = x_1...x_{j-1}1x_{j+1}...x_k0^{j-1}$ und $yz = y_1...y_{j-1}0y_{j+1}...y_k0^{j-1}$

und daher $xz \in L_k$ und $yz \notin L_k$.

Dies ist ein Widerspruch! Folglich gilt $\hat{\delta}_k(q_{0k}, x) \neq \hat{\delta}_k(q_{0k}, y)$ für alle paarweise unterschiedliche $x, y \in S_k = (\Sigma_{bool})^k$.

Daher hat B_k mindestens $|S_k| = 2^k$ viele Zustände.

■

6.4 Mindestanzahl Zustände

Die Grundidee ist es n Wörter anzugeben und zu beweisen, dass jedes von diesen n Wörtern in einem eigenen Zustand enden muss.

Seien $w_1, ..., w_n$ diese Wörter. Dann geben wir für jedes Paar von Wörtern $w_i \neq w_j$ einen Suffix $z_{i,j}$ an, so dass folgendes gilt:

$$w_iz_{i,j} \in L \iff w_jz_{i,j} \in L$$

Dann folgt aus Lemma 3.3

$$\hat{\delta}(q_0, w_i) \neq \hat{\delta}(q_0, w_j)$$

	w_2	...	w_n
w_1	$z_{1,2}$	...	$z_{1,n}$
...		...	...
w_{n-1}			$z_{n-1,n}$

Es eignet sich die Suffixe als Tabelle anzugeben.

Um die Wörter und Suffixe zu finden, kann es sich als nützlich erweisen, den Endlichen Automaten zu konstruieren.

Beweisschema

Wir nehmen zum Widerspruch an, dass es einen EA für L gibt mit weniger als n Zuständen.

Betrachten wir $w_1, ..., w_n$. Per Pigeonhole-Principle existiert $i < j$, so dass

$$\hat{\delta}(q_0, w_i) = \hat{\delta}(q_0, w_j)$$

Per Lemma 3.3 folgt daraus, dass

$$\forall z \in \Sigma^* : w_i z \in L \iff w_j z \in L$$

Für $z = z_{i,j}$ gilt aber per Tabelle

$$w_i z_{i,j} \in L \not\iff w_j z_{i,j} \in L \quad (\mathbf{1})$$

für alle $i < j$.

Da keines der n Wörter im gleichen Zustand enden kann, ergibt sich ein Widerspruch.

Dann noch Angabe der Tabelle für (**1**)

- Wenn es offensichtlich ist, muss (**1**) nicht bei jedem Suffix begründet werden.
- Ein minimaler endlicher Automat ist nicht notwendig für den Beweis. Hilft aber fürs

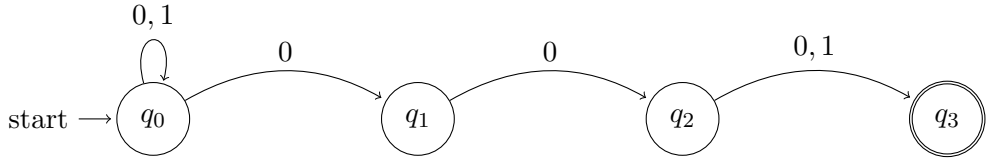
  i. Finden der w_i
  ii. Finden der $z_{i,j}$
  iii. Beweis von $w_i z_{i,j} \in L \not\iff w_j z_{i,j} \in L$ (Leicht überprüfbar)

Klassische Aufgabe - HS19 Aufgabe 3.a

Wir betrachten die Sprache

$$L = \{x00y \mid x \in \{0,1\}^* \text{ und } y \in \{0,1\}\}$$

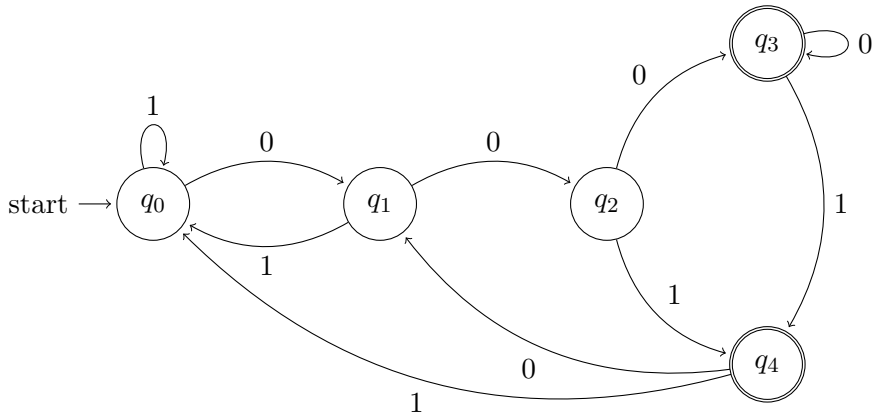
Konstruieren Sie einen nichtdeterminstischen endlichen Automaten mit höchstens 4 Zuständen, der L akzeptiert.



Klassische Aufgabe - HS19 Aufgabe 3.b

Zeigen Sie, dass jeder deterministische endliche Automat, der L akzeptiert, mindestens 5 Zustände braucht.

Wir zeichnen den zugehörigen EA zuerst.



Nehmen wir zum Widerspruch an, dass es einen endlichen Automaten gibt, der L akzeptiert und weniger als 4 Zustände hat.

Wir wählen die Wörter $B = \{\lambda, 0, 00, 000, 001\}$.

Nach dem Pigeonhole-Principle existieren zwei Wörter $w_i, w_j \in B, w_i \neq w_j$, so dass

$$\hat{\delta}(q_0, w_i) = \hat{\delta}(q_0, w_j)$$

Per Lemma 3.3 folgt daraus, dass

$$\forall z \in \Sigma^* : w_i z \in L \iff w_j z \in L$$

Wir betrachten folgende Tabelle mit Suffixen. Der zeigt für jedes Wortpaar $x, y \in B, x \neq y$ die Existenz

	0	00	000	001
λ	01	1	λ	λ
0		1	λ	λ
00			λ	λ
000				1

eines Suffixes z, so dass

$$(xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L)$$

Dies kann man mit den angegebenen Suffixen und dem angegebenen EA einfach überprüfen.

Dies widerspricht der vorigen Aussage, dass ein Wortpaar $w_i, w_j \in B, w_i \neq w_j$ existiert, so dass

$$\forall z \in \Sigma^* : w_i z \in L \iff w_j z \in L$$

Somit ist unsere Annahme falsch und ein EA für L muss mindestens 4 Zustände haben.

■

Bemerkung

Manchmal ist es zu schwierig einen minimalen EA zu finden und es funktioniert einfacher die Wörter durch Trial and Error zu finden. (Siehe Midterm HS22)

7 Turing Maschinen

7.1 Motivation und Überblick

Formalisierung notwendig, um mathematisch über die automatische Unlösbarkeit zu argumentieren.

Jede vernünftige Programmiersprache ist eine zulässige Formalisierung.

Aber nicht geeignet (meistens komplexe Operationen).

Die Turingmaschine erlaubt ein paar **elementare Operationen** und besitzt trotzdem die **volle Berechnungsstärke** beliebiger Programmiersprachen.

Ziel dieses Kapitels ist, dass ihr ein gewisse Gespür dafür bekommt, was eine Turingmaschine kann und was nicht.

7.2 Turing Maschinen - Formalisierung von Algorithmen

Informell

Eine Turingmaschine besteht aus

(i) einer endlichen Kontrolle, die das Programm enthält,

(ii) einem unendlichen Band, das als Eingabeband, aber auch als Speicher (Arbeitsband) zur Verfügung steht, und

(iii) einem Lese-/Schreibkopf, der sich in beiden Richtungen auf dem Band bewegen kann.

Für formale Beschreibung siehe Buch.

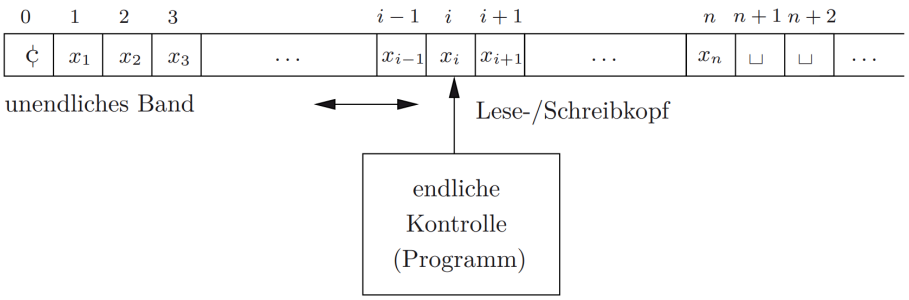


Figure 5: Abb. 4.1 vom Buch

Elementare Operation einer TM - Informell

Input

- Zustand der Maschine (der Kontrolle)
- Symbol auf dem Feld unter dem Lese-/Schreibkopf

Aktion

(i) ändert Zustand

(ii) schreibt auf das Feld unter dem Lese-/Schreibkopf

(iii) bewegt den Lese-/Schreibkopf nach links, rechts oder gar nicht. Ausser wenn ¢, dann ist links nicht möglich.

Eine **Konfiguration** C von M ist ein Element aus

$$\textbf{Konf(M)} = \{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+ \cup Q \cdot \{¢\} \cdot \Gamma^*$$

- Eine Konfiguration $¢w_1qaw_2$ mit $w_1, w_2 \in \Gamma^*$, $a \in \Gamma$ und $q \in Q$ sagt uns:

M im Zustand q, Inhalt des Bandes $¢w_1aw_2␣␣...$, Kopf an Position $|w_1|+1$ und liest gerade a.

- Eine Konfiguration $p¢w$ mit $p \in Q$, $w \in \Gamma^*$: Inhalt des Bandes $¢w␣␣...$, Zustand p und Kopf an Position 0.

Bmk: Im Buch haben sie in der Definition von Konf Γ^+ anstatt Γ^* an "letzter Stelle".

Es gibt wieder eine Schrittrelation $\vdash_M \subseteq \text{Konf}(M) \times \text{Konf}(M)$.

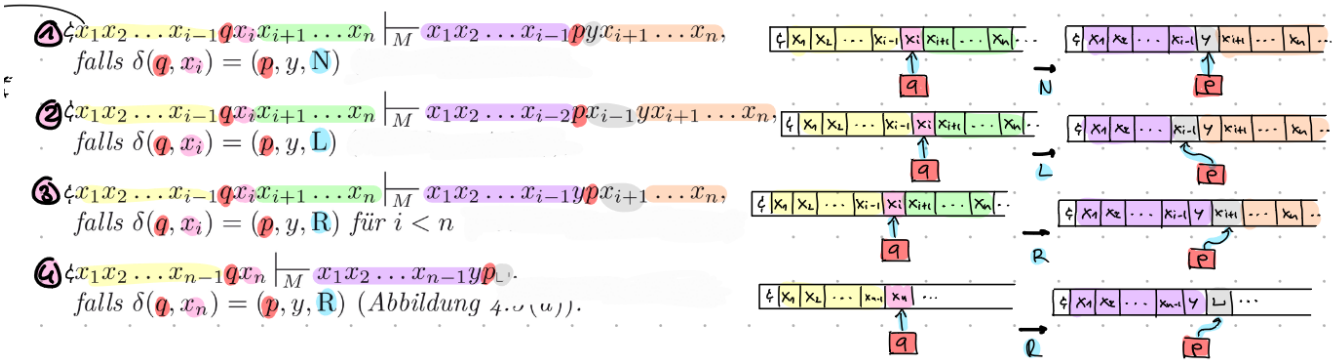


Figure 6: Diagramm von Adeline

Berechnung von M, **Berechnung von M auf einer Eingabe x** etc. durch $\vdash_M$ definiert.

Die Berechnung von M auf x heisst

- **akzeptierend**, falls sie in einer akzeptierenden Konfiguration $w_1 q_{\text{accept}} w_2$ endet (wobei ¢ in w_1 enthalten ist).
- **verwerfend**, wenn sie in in einer verwerfenden Konfiguration $w_1 q_{\text{reject}} w_2$ endet.
- **nicht-akzeptierend**, wenn sie entweder eine **verwerfende** oder unendliche Berechnung ist.

Die von der **Turingmaschine M akzeptierte Sprache** ist

$$\mathbf{L(M)} = \{w \in \Sigma^* \mid q_0 ¢ w \vdash_M^* y q_{\text{accept}} z, \text{ für irgendwelche } y, z \in \Gamma^*\}$$

7.3 Wichtige Klassen

Reguläre Sprachen

$$\mathcal{L}_{\mathbf{EA}} = \{L(A) \mid A \text{ ist ein EA}\} = \mathcal{L}_{\mathbf{NEA}}$$

Rekursiv aufzählbare Sprachen

Eien Sprache $L \subseteq \Sigma^*$ heisst **rekursiv aufzählbar**, falls eine TM M existiert, so dass $L = L(M)$.

$$\mathcal{L}_{\mathbf{RE}} = \{L(M) \mid M \text{ ist eine TM}\}$$

ist die **Klasse aller rekursiv aufzählbaren Sprachen**.

Halten

Wir sagen das M **immer hält**, wenn für alle Eingaben $x \in \Sigma^*$

(i) $q_0 ¢ x \vdash_M^* y q_{\text{accept}} z, y, z \in \Gamma^*$, falls $x \in L$ und

(ii) $q_0 ¢ x \vdash_M^* u q_{\text{reject}} v, u, v \in \Gamma^*$, falls $x \notin L$.

Rekusive Sprachen

Eine Sprache $L \subseteq \Sigma^*$ heisst **rekursiv (entscheidbar)**, falls $L = L(M)$ für eine TM M, die **immer hält**.

$$\mathcal{L}_{\mathbf{R}} = \{L(M) \mid M \text{ ist eine TM, die immer hält}\}$$

ist die **Klasse der rekursiven (algorithmisch erkennbaren) Sprachen**.

7.4 Mehrband-Turingmaschine

Mehrband-TM - Informelle Beschreibung

Für $k \in \mathbb{N} \setminus \{0\}$ hat eine k-Band Turingmaschine

- eine endliche Kontrolle
- ein endliches Band mit einem Lesekopf (Eingabeband)
- k Arbeitsbänder, jedes mit eigenem Lese-/Schreibkopf (nach rechts unendlich)

Insbesondere gilt 1-Band TM $\neq$ "normale" TM

Am Anfang der Berechnung einer MTM M auf w

- Arbeitsbänder "leer" und die k Lese-/Schreibköpfe auf Position 0.
- Inhalt des Eingabebands $¢w\$$ und Lesekopf auf Position 0.
- Endliche Kontrolle im Zustand q_0.

7.5 Äquivalenz von Maschinen (TM, MTM)

Seien A und B zwei Maschinen mit **gleichem** Σ.
Wir sagen, dass **A äquivalent zu B ist**, wenn für jede Eingabe $x \in \Sigma^*$

(i) A akzeptiert $x \iff B$ akzeptiert x

(ii) A verwirft $x \iff B$ verwirft x

(iii) A arbeitet unendlich lange auf $x \iff B$ arbeitet unendlich lange auf x

Wir haben

$$A \text{ und } B \text{ äquivalent} \implies L(A) = L(B)$$

aber

$$L(A) = L(B) \not\implies A \text{ und } B \text{ äquivalent}$$

da A auf x unendlich lange arbeiten könnte, während B x verwirft.

Lemma 4.1

Zu jeder TM A existiert eine zu A äquivalente 1-Band-TM B

Beweisidee B kopiert die Eingabe zuerst aufs Arbeitsband und simuliert dann A.

Lemma 4.2

Zu jeder Mehrband-TM A existiert eine zu A äquivalente TM B

Beweis

Sei A eine k-Band-Turingmaschine für ein $k \in \mathbb{N} \setminus \{0\}$. Wir konstruieren eine TM B, die Schritt für Schritt A simuliert.

B speichert die Inhalte aller $k+1$ Bänder von A auf ihrem einzigen Band. Anschaulich gesprochen ist jedes Feld auf dem Band von B ein $2(k+1)$-Tupel und jedes Element dieses Tupels ist auf einer Spur. Sei Γ_A das Arbeitsalphabet von A. Dann gilt

$$\Gamma_B = (\Sigma_A \cup \{¢, \$, \textvisiblespace\}) \times \{\textvisiblespace, \uparrow\} \times (\Gamma_A \times \{\textvisiblespace, \uparrow\})^k \cup \Sigma_A \cup \{\textvisiblespace, ¢\}$$

Für ein Symbol $\alpha = (a_0, a_1, a_2, ..., a_{2k+1}) \in \Gamma_B$ sagen wir, dass a_i auf der i-ten Spur liegt. Daher bestimmen die i-ten Elemente der Symbole auf dem Band von B den Inhalt der i-ten Spur. Eine Konfiguration $(q, w, i, x_1, i_1, x_2, i_2, ..., x_k, i_k)$ von A ist dann in B wie folgt gespeichert.

- Der Zustand q ist in der endlichen Kontrolle von B gespeichert.
- Die 0-te Spur des Bandes von B enthält die $¢w\$$ (i.e. den Inhalt des Eingabebandes von A)
- Für alle $i \in \{1, ..., k\}$ enthält die $(2i)$-te Spur des Bandes von B den Inhalt vom i-ten Band von A (i.e. $¢x_i\$$).
- Für alle $i \in \{1, ..., k\}$ bestimmt die $(2i+1)$-te Spur des Bandes von B mit dem Symbol $\uparrow$ die Position des Kopfes auf dem i-ten Arbeitsband von A.

Ein Schritt von A kann jetzt durch folgende Prozedur von B simuliert werden:

1. B liest einmal den Inhalt ihres Bandes von links nach rechts, bis sie alle $k+1$ Kopfpositionen von A gefunden hat, und speichert dabei in ihrem Zustand die $k+1$ Symbole, die an diesen Positionen stehen. (Dies kann ohne weiteres in der Zustandsmenge abgespeichert werden, da k fix ist, folglich ist dann Γ_A^k auch endlich)
2. Nach der ersten Phase kennt B das ganze Argument (der Zustand von A ist im Zustand von B gespeichert) der Transitionsfunktion von A und kann also die entsprechenden Aktionen (Köpfe bewegen, Ersetzen von Symbolen) von A bestimmen. Diese Änderungen führt B in einem Lauf über ihr Band von rechts nach links durch.

■

Aus Lemma 4.1 und 4.2 folgt direkt

Satz 4.1

Die Maschinenmodelle von Turingmaschinen und Mehrband-Turingmaschinen sind äquivalent.

Note:

- "Äquivalenz" für Maschinenmodelle wird in Definition 4.2 definiert.
- Maschinenmodelle sind Klassen von Maschinen (i.e. Mengen von Maschinen mit gewissen Eigenschaften).

7.6 Nichtdeterministische Turingmaschinen

Definition von NTM

Eine **nichtdeterministische Turingmaschine (NTM)** ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, wobei

(i) $Q, \Sigma, \Gamma, q_{\text{accept}}, q_{\text{reject}}$ die gleiche Bedeutung wie bei einer TM haben, und

(ii) $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L, R, N}\})$ die **Übergangsfunktion** von M ist und die folgende Eigenschaft hat:

$$\delta(p, ¢) \subseteq \{(q, ¢, X) \mid q \in Q, X \in \{R, N\}\}$$

für alle $p \in Q$

Konfiguration ähnlich wie bei TMs.

$$\text{Konfiguration akzeptierend} \iff \text{enthält } q_{\text{accept}}$$
$$\text{Konfiguration verwerfend} \iff \text{enthält } q_{\text{reject}}$$

Die üblichen Sachen

- Schrittrelation $\vdash_M$ "verbindet zwei Konfigurationen, wenn man von der einen in die andere kommen kann"
- Reflexive und transitive Hülle ist $\vdash_M^*$.
- Berechnung von M ist eine Folge von Konfigurationen $C_1, C_2, ...$, so dass $C_i \vdash_M C_{i+1}$.
- Eine Berechnung von M auf x ist beginnt in $q_0¢x$ und endet entweder unendlich oder endet in $\{q_{\text{accept}}, q_{\text{reject}}\}$.

Akzeptierte Sprache

$$L(M) = \{w \in \Sigma^* \mid q_0¢w \vdash_M^* yq_{\text{accept}}z \text{ für irgendwelche } y, z \in \Gamma^*\}$$

Berechnungsbaum

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ eine NTM und sei x ein Wort über dem Eingabealphabet Σ von M. Ein **Berechnungsbaum $T_{M,x}$ von M auf x** ist ein (potentiell unendlicher) gerichteter Baum mit einer Wurzel, der wie folgt definiert wird.

(i) Jeder Knoten von $T_{M,x}$ ist mit einer Konfiguration beschriftet.

(ii) Die Wurzel ist der einzige Knoten von $T_{M,x}$ mit dem Eingangsgrad 0 und ist mit der Startkonfiguration $q_0¢x$ beschriftet.

(iii) Jeder Knoten des Baumes, der mit einer Konfiguration C beschriftet ist, hat genauso viele Kinder wie C Nachfolgekonfigurationen hat, und diese Kinder sind mit diesen Nachfolgekonfigurationen C markiert.

Äquivalenz NTM und TM

Satz 4.2

Sei M eine NTM. Dann existiert eine TM A, so dass

(i) $L(M) = L(A)$ und

(ii) falls M keine unendlichen Berechnungen auf Wörtern aus $L(M)^{\complement}$ hat, dann hält A immer.

Beweisidee:

"BFS im Berechnungsbaum", i.e. wir simulieren einzelne Schritte der verschiedenen Berechnungsstränge.

8 Einstieg Berechnenbarkeit

8.1 Diagonalisierung

Bijektion, Injektion, Schreibweise

Seien A und B zwei Mengen.
Wir sagen, dass

i. $|\mathbf{A}| \leq |\mathbf{B}|$, falls eine injektive Funktion $f : A \to B$ existiert.

ii. $|\mathbf{A}| = |\mathbf{B}|$, falls $|A| \leq |B|$ und $|B| \leq |A|$.

iii. $|\mathbf{A}| < |\mathbf{B}|$, falls $|A| \leq |B|$ und keine injektive Abbildung von B nach A existiert.

Zur Erinnerung:

$$f : A \to B \text{ injektiv } \iff \forall x, y \in A, x \neq y . f(x) \neq f(y)$$

Abzählbarkeit

E

ine Menge A heisst **abzählbar**, falls A endlich ist oder $|A| = |\mathbb{N}|$.

Lemma 5.1

Sei Σ ein beliebiges Alphabet. Dann ist Σ^* abzählbar.

Satz 5.1

Die Menge **KodTM** der Turingmaschinenkodierungen ist abzählbar.

Beweisidee

KodTM $\subseteq (\Sigma_{\text{bool}})^*$ und Lemma 5.1

Lemma 5.2

$(\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ ist abzählbar.

Beweisidee

Unendliche 2-dimensionale Tabelle, so dass an der i-ten Zeile und j-ten Spalte, sich das Element $(i, j) \in (\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ befindet.

Formal definiert man dabei die lineare Ordnung

$$(a, b) < (c, d) \iff a + b < c + d \text{ oder } (a + b = c + d \text{ und } b < d)$$

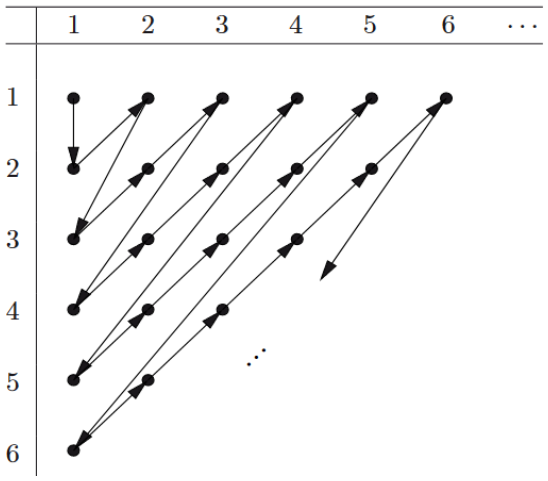


Figure 7: Abbildung 5.3 im Buch

Die i-te Diagonale hat i Elemente. Ein beliebiges Element $(a, b) \in (\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ ist das b-te Element auf der $(a + b - 1)$-ten Diagonale.

Auf den ersten $a + b - 2$ Diagonalen gibt es

$$\sum_{i=1}^{a+b-2} i = \frac{(a+b-2) \cdot ((a+b-2)+1)}{2} = \binom{a+b-1}{2}$$

Elemente.

Folglich ist

$$f((a, b)) = \binom{a+b-1}{2} + b$$

eine Bijektion von $(\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ nach $\mathbb{N} \setminus \{0\}$.

Überabzählbarkeit

Satz 5.3

$[0, 1]$ ist nicht abzählbar.

Beweisidee

Klassisches Diagonalisierungsargument. Aufpassen auf 0 und 9. I.e. $1 = 0.\overline{99}$.

$f(x)$	$x \in [0,1]$							
1	0.	$\boxed{a_{11}}$	a_{12}	a_{13}	a_{14}	⋯		
2	0.	a_{21}	$\boxed{a_{22}}$	a_{23}	a_{24}	⋯		
3	0.	a_{31}	a_{32}	$\boxed{a_{33}}$	a_{34}	⋯		
4	0.	a_{41}	a_{42}	a_{43}	$\boxed{a_{44}}$	⋯		
⋮	⋮	⋮	⋮	⋮		⋯		
i	0.	a_{i1}	a_{i2}	a_{i3}	a_{i4}	⋯	$\boxed{a_{ii}}$	⋯
⋮	⋮							

Abbildung 5.5

Satz 5.4

$\mathcal{P}((\Sigma_{\text{bool}})^*)$ ist nicht abzählbar.

Beweis:

Wir definieren eine injektive Funktion von $f : [0,1] \to \mathcal{P}((\Sigma_{bool})^*)$ und beweisen so $|\mathcal{P}((\Sigma_{bool})^*)| \geq |[0,1]|$.

Sei $a \in [0,1]$ beliebig. Wir können a wie folgt binär darstellen:

$$\text{Nummer}(a) = 0.a_1a_2a_3a_4... \text{ mit } a = \sum_{i=1}^{\infty} a_i \cdot 2^{-i}.$$

Hier ist zu beachten, dass wir für eine Zahl a immer die lexikographisch letzte Darstellung wählen.

Dies tun wir, weil eine reelle Zahl 2 verschiedene Binärdarstellungen haben kann. Beispiel: $\frac{1}{2} = 0.1\overline{0} = 0.0\overline{1}$.

Für jedes a definieren wir:

$$f(a) = \{a_1, a_2a_3, a_4a_5a_6, ..., a_{\binom{n}{2}+1}a_{\binom{n}{2}+2}...a_{\binom{n+1}{2}}, ...\}$$

Da $f(a) \subseteq (\Sigma_{bool})^*$ gilt $f(a) \in \mathcal{P}((\Sigma_{bool})^*)$.

Wir haben für alle $n \in \mathbb{N} \setminus \{0\}$, dass $f(a)$ **genau** ein Wort dieser Länge enthält. Nun können wir daraus folgendes schliessen:

Weil die Binärdarstellung zweier unterschiedlichen reellen Zahlen an mindestens einer Stelle unterschiedlich ist, gilt $b \neq c \implies f(b) \neq f(c), \forall b, c \in [0,1]$.

Folglich ist f injektiv und wir haben $|\mathcal{P}((\Sigma_{bool})^*)| \geq |[0,1]|$.

Da $[0,1]$ nicht abzählbar ist, folgt daraus:

$\mathcal{P}((\Sigma_{bool})^*)$ ist nicht abzählbar.

■

Diagonalsprache L_{diag}

Zur Erinnerung:

Rekursiv aufzählbare Sprachen

Eien Sprache $L \subseteq \Sigma^*$ heisst **rekursiv aufzählbar**, falls eine TM M existiert, so dass $L = L(M)$.

$$\mathcal{L}_{\text{RE}} = \{L(M) \mid M \text{ ist eine TM}\}$$

ist die **Klasse aller rekursiv aufzählbaren Sprachen**.

Wir zeigen jetzt per Diagonalisierung, die Existenz einer Sprache die nicht rekursiv aufzählbar ist.

Sei $w_1, w_2, ...$ die kanonische Ordnung aller Wörter über Σ_{bool} und sei $M_1, M_2, M_3, ...$ die Folge aller Turingmaschinen.

Wir definieren eine unendliche (bool'sche) Matrix $A = [d_{ij}]_{i,j=1,2,...}$ mit

$$d_{ij} = 1 \iff M_i \text{ akzeptiert } w_j.$$

Wir definieren

$$L_{\text{diag}} = \{w \mid w = w_i \text{ und } M_i \text{ akzeptiert } w_i \text{ nicht für ein } i \in \mathbb{N} \setminus \{0\}\}$$

Satz 5.5

$$L_{\text{diag}} \notin \mathcal{L}_{\text{RE}}$$

Beweis:

Wir haben

$$L_{\text{diag}} = \{w \mid w = w_i \text{ und } M_i \text{ akzeptiert } w_i \text{ nicht für ein } i \in \mathbb{N} \setminus \{0\}\}$$

Widerspruchsbeweis:

Sei $L_{\text{diag}} \in \mathcal{L}_{\text{RE}}$. Dann existiert eine TM M, so dass $L(M) = L_{\text{diag}}$. Da diese TM eine TM in der Nummerierung aller TM ist, existiert ein $i \in \mathbb{N}$, so dass $M_i = M$.

Wir betrachten nun das Wort w_i für diese $i \in \mathbb{N}$. Per Definition von L_{diag}, gilt:

$$w_i \in L_{\text{diag}} \iff w_i \notin L(M_i)$$

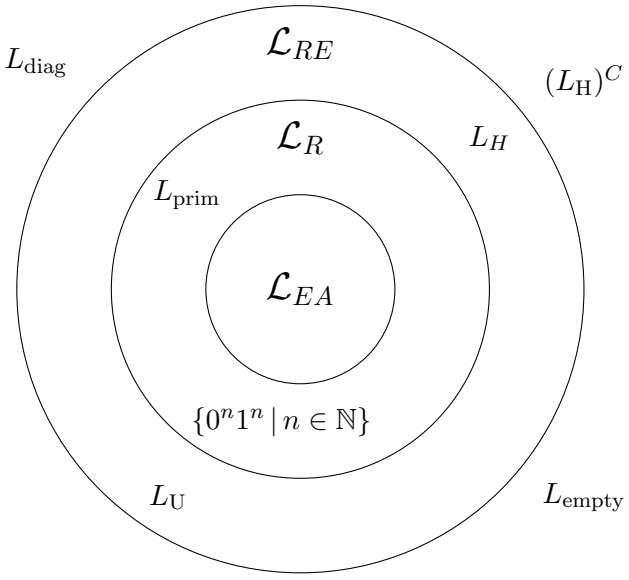
Da aber $L(M_i) = L_{\text{diag}}$, haben wir folgenden Widerspruch:

$$w_i \in L_{\text{diag}} \iff w_i \notin L_{\text{diag}}$$

Folglich gilt $L_{\text{diag}} \notin \mathcal{L}_{\text{RE}}$.

■

8.2 Klassifizierung verschiedener Sprachen



8.3 Begrifflichkeiten

Für eine Sprache L gilt folgendes

$$\begin{gathered} L \text{ regulär} \iff L \in \mathcal{L}_{\mathbf{EA}} \iff \exists \text{ EA } A \text{ mit } L(A) = L \\ L \text{ rekursiv} \iff L \in \mathcal{L}_{\mathbf{R}} \iff \exists \text{ Alg. A mit } L(A) = L \\ L \text{ rekursiv aufzählbar} \iff L \in \mathcal{L}_{\mathbf{RE}} \iff \exists \text{ TM } M.\ L(M) = L \end{gathered}$$

- "Algorithmus" = TM, die immer hält.
- L rekursiv = L entscheidbar
- L rekursiv aufzählbar = L erkennbar

9 Reduktion

- Reduktionen sind klassische Aufgaben an dem Endterm. Ein bisschen wie Nichtregularitätsbeweise.
- Ist aber auch nicht so schlimm.

9.1 R-Reduktion

Definition 5.3

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Wir sagen, dass $\mathbf{L_1}$ **auf** $\mathbf{L_2}$ **rekursiv reduzierbar ist,** $\mathbf{L_1} \leq_{\mathbf{R}} \mathbf{L_2}$, falls

$$L_2 \in \mathcal{L}_{\mathrm{R}} \implies L_1 \in \mathcal{L}_{\mathrm{R}}$$

Bemerkung:

Intuitiv bedeutet das *"L_2 mindestens so schwer wie L_1"* (bzgl. algorithmischen Lösbarkeit).

9.2 EE-Reduktion

Definition 5.4

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Wir sagen, dass $\mathbf{L_1}$ **auf** $\mathbf{L_2}$ **EE-reduzierbar ist,** $\mathbf{L_1} \leq_{\mathbf{EE}} \mathbf{L_2}$, wenn eine TM M existiert, die eine Abbildung $f_M : \Sigma_1^* \to \Sigma_2^*$ mit der Eigenschaft

$$x \in L_1 \iff f_M(x) \in L_2$$

für alle $x \in \Sigma_1^*$ berechnet. Wir sagen auch, dass die TM M die Sprache L_1 auf die Sprache L_2 reduziert.

Wir sagen, dass M eine Funktion $F : \Sigma^* \to \Gamma^*$ **berechnet**, falls für alle $x \in \Sigma^*$: $q_0 ¢ x \vdash_M^* q_{\text{accept}} ¢ F(x)$.

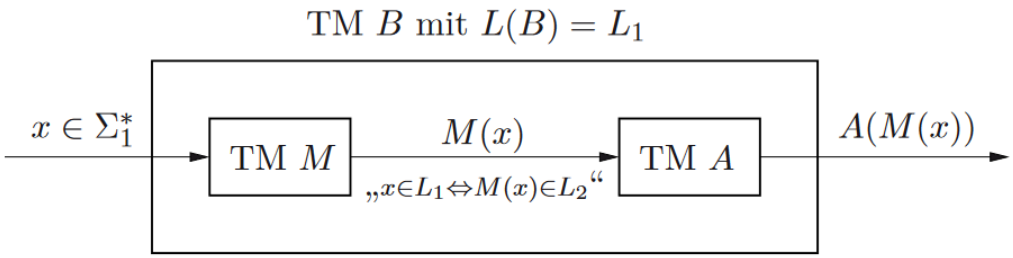


Figure 8: Abbildung 5.7 vom Buch

9.3 Verhältnis von EE-Reduktion und R-Reduktion

Lemma 5.3

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen.

$$L_1 \leq_{\mathrm{EE}} L_2 \implies L_1 \leq_{\mathrm{R}} L_2$$

Beweis:

$$L_1 \leq_{\mathrm{EE}} L_2 \implies \exists \text{TM } M.\ x \in L_1 \iff M(x) \in L_2$$

Wir zeigen nun $L_1 \leq_R L_2$, i.e. $L_2 \in \mathcal{L}_R \implies L_1 \in \mathcal{L}_R$.

Sei $L_2 \in \mathcal{L}_R$. Dann existiert ein Algorithmus A (TM, die immer hält), der L_2 entscheidet.

Wir konstruieren eine TM B (die immer hält) mit $L(B) = L_1$

Für eine Eingabe $x \in \Sigma_1^*$ arbeitet B wie folgt:

(i) B simuliert die Arbeit von M auf x, bis auf dem Band das Wort $M(x)$ steht.

(ii) B simuliert die Arbeit von A auf $M(x)$.
Wenn A das Wort $M(x)$ akzeptiert, dann akzeptiert B das Wort x.
Wenn A das Wort $M(x)$ verwirft, dann verwirft B das Wort x.

A hält immer $\implies$ B hält immer und somit gilt $L_1 \in \mathcal{L}_R$

■

9.4 L und $L^{\complement}$

Lemma 5.4

Sei Σ ein Alphabet. Für jede Sprache $L \subseteq \Sigma^*$ gilt:

$$L \leq_R L^{\complement} \text{ und } L^{\complement} \leq_R L$$

Beweis:

Es reicht $L^{\complement} \leq_R L$ zu zeigen, da $(L^{\complement})^{\complement} = L$ und somit dann $(L^{\complement})^{\complement} = L \leq_R L^{\complement}$.

Sei M' ein Algorithmus für L, der immer hält ($L \in \mathcal{L}_R$). Dann beschreiben wir einen Algorithmus B, der $L^{\complement}$ entscheidet.

B übernimmt die Eingaben und gibt sie an M' weiter und invertiert dann die Entscheidung von M'. Weil M' immer hält, hält auch B immer und wir haben offensichtlich $L(B) = L$.

■

Korollar 5.2

$$(L_{\text{diag}})^{\complement} \notin \mathcal{L}_R$$

Beweis:

Aus Lemma 5.4 haben wir $L_{\text{diag}} \leq_R (L_{\text{diag}})^{\complement}$. Daraus folgt $L_{\text{diag}} \notin \mathcal{L}_R \implies (L_{\text{diag}})^{\complement} \notin \mathcal{L}_R$. Da $L_{\text{diag}} \notin \mathcal{L}_{RE}$ gilt auch $L_{\text{diag}} \notin \mathcal{L}_R$.

Folglich gilt $(L_{\text{diag}})^{\complement} \notin \mathcal{L}_R$.

■

Lemma 5.5

$$L_{\text{diag}}^{\complement} \in \mathcal{L}_{RE}$$

Beweis

Direkter Beweis: Wir beschreiben eine TM A mit $L(A) = L_{\text{diag}}^{\complement}$.

Eingabe: $x \in (\Sigma_{\text{bool}})^*$

(i) Berechne i, so dass $w_i = x$ in kanonischer Ordnung

(ii) Generiere $\text{Kod}(M_i)$.

(iii) Simuliere die Berechnung von M_i auf $w_i = x$.

- Falls $w_i \in L(M_i)$ akzeptiert, akzeptiert A die Eingabe x.
- Falls M_i verwirft (hält in q_{reject}), dann hält A und verwirft $x = w_i$ auch.
- Falls M_i unendlich lange arbeitet, wird A auch nicht halten und dann folgt auch $x \notin L(A)$.

Aus dem folgt $L(A) = L_{\text{diag}}^{\complement}$.

■

Korollar 5.3

$L_{\text{diag}}^{\complement} \in \mathcal{L}_{\text{RE}} \setminus \mathcal{L}_{\text{R}}$ und daher $\mathcal{L}_{\text{R}} \subsetneq \mathcal{L}_{\text{RE}}$.

9.5 Universelle Sprache

Sei

$$L_{\text{U}} := \{\text{Kod}(M)\#w \mid w \in (\Sigma_{\text{bool}})^* \text{ und } M \text{ akzeptiert } w\}$$

Satz 5.6

Es gibt eine TM U, **universelle TM** genannt, so dass

$$L(U) = L_{\text{U}}$$

Daher gilt $L_{\text{U}} \in \mathcal{L}_{\text{RE}}$.

Beweis

Direkter Beweis: Konstruktion einer TM.

Siehe Buch/Vorlesung.

Satz 5.7

$$L_{\text{U}} \notin \mathcal{L}_{\text{R}}$$

Beweis

Wir zeigen $L_{\text{diag}}^{\complement} \leq_{\text{R}} L_{\text{U}}$.

Siehe Buch/Vorlesung.

9.6 Halteproblem

Sei

$$L_{\text{H}} = \{\text{Kod}(M)\#w \mid M \text{ hält auf } w\}$$

Satz 5.8

$$L_{\text{H}} \notin \mathcal{L}_{\text{R}}$$

Beweis

Wir zeigen $L_{\text{U}} \leq_{\text{R}} L_{\text{H}}$.

Siehe Buch/Vorlesung.

9.7 Parallele Simulation vs Nichtdeterminismus

Sei

$$L_{\text{empty}} = \{\text{Kod}(M) \mid L(M) = \emptyset\}$$

und

$$L_{\text{empty}}^{\complement} = \{\text{Kod}(M) \mid L(M) \neq \emptyset\} \cup \{x \in \{0,1,\#\} \mid x \notin \textbf{KodTM}\}$$

Lemma 5.6

$$L_{\text{empty}}^{\complement} \in \mathcal{L}_{\text{RE}}$$

Nichtdeterminismus

Beweis

Da für jede NTM M_1 eine TM M_2 existiert mit $L(M_1) = L(M_2)$, reicht es eine NTM M_1 mit $L(M_1) = L_{\text{empty}}^{\complement}$ zu finden.

Eingabe: $x \in \{0,1,\#\}$

(i) M_1 prüft deterministisch, ob $x = \text{Kod}(M)$ für eine TM M.
Falls x keine TM kodiert, wird x akzeptiert.

(ii) Sonst gilt $x = \text{Kod}(M)$ für eine TM M und M_1 wählt nichtdeterministisch ein Wort $y \in (\Sigma_{\text{bool}})^*$.

(iii) Dann simuliert M_1 die TM M auf y deterministisch und übernimmt die Ausgabe.

Wir unterscheiden zwischen 3 Fällen

I $x = \text{Kod}(M)$ und $L(M) = \emptyset$
Dann gilt $x \notin L_{\text{empty}}^{\complement}$ und da es keine akzeptierende Berechnung gibt, auch $x \notin L(M_1)$.

II $x = \text{Kod}(M)$ und $L(M) \neq \emptyset$
Dann gilt $x \in L_{\text{empty}}^{\complement}$ und da es eine akzeptierende Berechnung gibt, auch $x \in L(M_1)$.

III x kodiert keine TM
Wir haben $x \in L_{\text{empty}}^{\complement}$ und wegen Schritt (i) auch $x \in L(M_1)$.

Somit gilt $L(M_1) = L_{\text{empty}}^{\complement}$.

■

Parallele Simulation

Alternativer Beweis

Wir konstruieren eine TM A mit $L(A) = L$ direkt. Eingabe: $x \in \{0,1,\#\}$

I. Falls x keine Kodierung einer TM ist, akzeptiert A die Eingabe.

II. Falls $x = \text{Kod}(M)$ für eine TM M, arbeitet A wie folgt

- Generiert systematisch alle Paare $(i,j) \in (\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$. (Abzählbarkeit)
- Für jedes Paar (i,j), generiert A das kanonisch i-te Wort w_i und simuliert j Berechnungsschritte der TM M auf w_i.
- Falls M an ein Wort akzeptiert, akzeptiert A das Wort x.

Falls $L(M) \neq \emptyset$ existiert ein $y \in L(M)$. Dann ist $y = w_k$ für ein $k \in \mathbb{N} \setminus \{0\}$ und die akzeptierende Berechnung von M auf y hat eine endliche Länge l.

Das Paar (k, l) wird in endlich vielen Schritten erreicht und somit akzeptiert A die Eingabe x, falls $L(M) \neq \emptyset$.

Somit folgt $L(A) = L^{\complement}_{\text{diag}}$.

■

9.8 Aufgabe 5.22

Wir zeigen

$$L \in \mathcal{L}_{\text{RE}} \wedge L^{\complement} \in \mathcal{L}_{\text{RE}} \iff L \in \mathcal{L}_{\text{R}}$$

$(\implies)$:

Nehmen wir $L \in \mathcal{L}_{\text{RE}} \wedge L^{\complement} \in \mathcal{L}_{\text{RE}}$ an.

Dann existiert eine TM M und M_C mit $L(M) = L$ und $L(M_C) = L^{\complement}$.

Wir konstruieren eine TM A, die für eine Eingabe w die beiden TM's M und M_C parallel auf w simuliert.

A akzeptiert w, falls M das Wort akzeptiert und verwirft, falls M_C das Wort akzeptiert.

Bemerke, dass $L(M) \cap L(M_C) = \emptyset$ und $L(M) \cup L(M_C) = \Sigma^*$.

Da $w \in L(M)$ oder $w \in L(M_C)$, hält A immer.

Da A genau dann akzeptiert, falls $w \in L(M)$, folgt $L(A) = L(M) = L$.

Demnach gilt $L \in \mathcal{L}_{\text{R}}$.

$(\impliedby)$:

Nehmen wir $L \in \mathcal{L}_{\text{R}}$ an. Per Lemma 5.4 gilt $L^{\complement} \leq_{\text{R}} L$ und daraus folgt auch $L^{\complement} \in \mathcal{L}_{\text{R}}$.

Da $\mathcal{L}_{\text{R}} \subset \mathcal{L}_{\text{RE}}$, folgt $L \in \mathcal{L}_{\text{RE}} \wedge L^{\complement} \in \mathcal{L}_{\text{RE}}$.

■

Lemma 5.7

$$L^{\complement}_{\text{empty}} \notin \mathcal{L}_{\text{R}}$$

Beweis

Wir zeigen $L_{\text{U}} \leq_{\text{EE}} L^{\complement}_{\text{empty}}$.

Siehe Buch/Vorlesung.

Korollar 5.4

$$L_{\text{empty}} \notin \mathcal{L}_{\text{R}}$$

Korollar 5.5

$$L_{\text{EQ}} \notin \mathcal{L}_{\text{R}}$$

für $L_{\text{EQ}} = \{\text{Kod}(M)\#\text{Kod}(\overline{M}) \mid L(M) = L(\overline{M})\}$.

9.9 Beispielaufgabe 17a HS22

Beweise

$$L_{\mathrm{H}} \leq_{\mathrm{EE}} L_{\mathrm{U}}$$

wobei

$$L_{\mathrm{H}} = \{\mathrm{Kod}(M)\#w \mid M \text{ hält auf } w \wedge w \in (\Sigma_{\mathrm{bool}})^*\}$$

und

$$L_{\mathrm{U}} = \{\mathrm{Kod}(M)\#w \mid M \text{ akzeptiert } w \wedge w \in (\Sigma_{\mathrm{bool}})^*\}$$

Lösung

Wir wollen $L_{\mathrm{H}} \leq_{\mathrm{EE}} L_{\mathrm{U}}$ zeigen.

Wir geben die Reduktion zuerst als Zeichnung an.

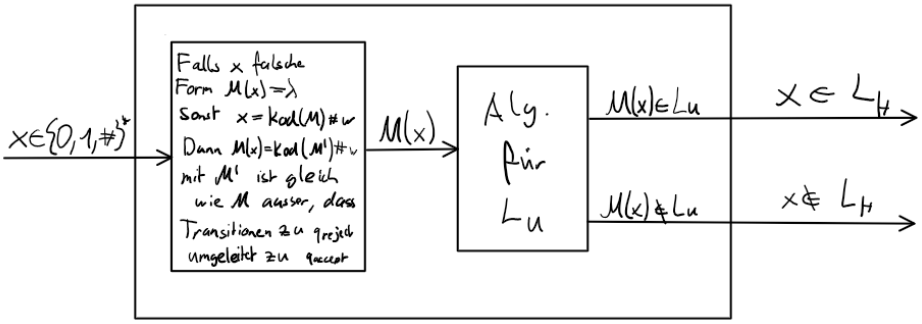


Figure 9: EE-Reduktion von L_{H} auf L_{U}

Wir definieren eine Funktion $M(x)$ für ein $x \in \{0, 1, \#\}^*$, so dass

$$x \in L_{\mathrm{H}} \iff M(x) \in L_{\mathrm{U}} \qquad (1)$$

Falls x nicht die richtige Form hat, ist $M(x) = \lambda$, sonst ist $M(x) = \mathrm{Kod}(M')\#w$ wobei M' gleich aufgebaut ist wie M, ausser dass alle Transitionen zu q_{reject} zu q_{accept} umgeleitet werden. Wir sehen, dass M' genau dann w akzeptiert, wenn M auf w hält.

Dieses $M(x)$ übergeben wir dem Algorithmus für L_{U}.

Wir beweisen nun $x \in L_{\mathrm{H}} \iff M(x) \in L_{\mathrm{U}}$:

(i) $x \in L_{\mathrm{H}}$

Dann ist $x = \mathrm{Kod}(M)\#w$ von der richtigen Form, und M hält auf w. Das heisst die Simulation von M auf w endet entweder in q_{reject} oder in q_{accept}.

Folglich wird M' w immer akzeptieren, da alle Transitionen zu q_{reject} zu q_{accept} umgeleitet wurden.

$x \in L_{\mathrm{H}} \implies M(x) \in L_{\mathrm{U}}$

(ii) $x \notin L_{\mathrm{H}}$

Dann unterscheiden wir zwischen zwei Fällen:

(a) x hat nicht die richtige Form, i.e. $x \neq \mathrm{Kod}(M)\#w$. Dann ist $M(x) = \lambda$ und da es keine Kodierung einer Turingmaschine M gibt, so dass $\mathrm{Kod}(M) = \lambda$, gilt $\lambda \notin L_{\mathrm{U}}$.

(b) $x = \text{Kod}(M)\#w$ hat die richtige Form. Dann haben wir $M(x) = \text{Kod}(M')\#w$.
Da aber $x \notin L_{\text{H}}$, hält M nicht auf w. Da M nicht auf w hält, erreicht es nie q_{reject} oder q_{accept} in M und so wird w von M' nicht akzeptiert.
$\implies M(x) \notin L_{\text{U}}$

So haben wir mit diesen Fällen (a) und (b) $x \notin L_{\text{H}} \implies M(x) \notin L_{\text{U}}$ bewiesen.

Aus indirekter Implikation folgt $M(x) \in L_{\text{U}} \implies x \in L_{\text{H}}$

Aus (i) und (ii) folgt

$$x \in L_{\text{H}} \iff M(x) \in L_{\text{U}} \qquad (1)$$

Somit ist die Reduktion korrekt.

■

9.10 Beispielaufgabe 18b HS22

Sei

$$L_{\text{infinite}} = \{\text{Kod}(M) \mid M \text{ hält auf keiner Eingabe}\}$$

Zeige $(L_{\text{infinite}})^C \notin \mathcal{L}_{\text{R}}$

Lösung

Wir zeigen, dass $(L_{\text{infinite}})^C \notin \mathcal{L}_{\text{R}}$ mit einer geeigneten Reduktion.

Wir beweisen $L_{\text{H}} \leq_{\text{R}} (L_{\text{infinite}})^C$

Um dies zu zeigen nehmen wir an, dass wir einen Algorithmus A haben, der $(L_{\text{infinite}})^C$ entscheidet. Wir konstruieren einen Algorithmus B, der mit Hilfe von A, die Sprache L_{H} entscheidet.

Wir betrachten folgende Abbildung:

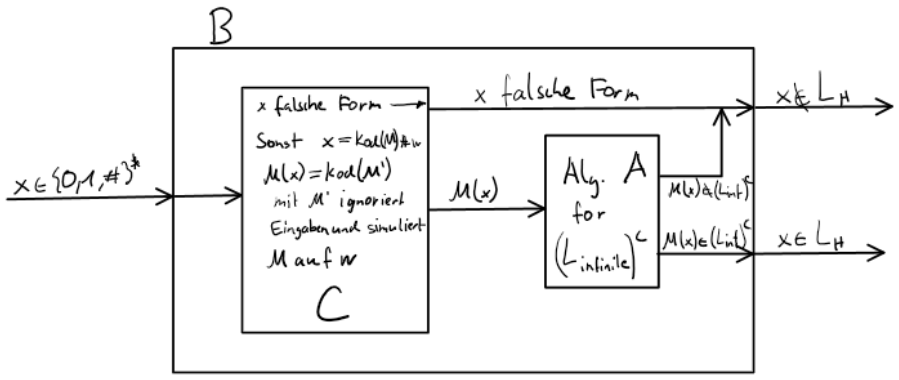


Figure 10: R-Reduktion von L_{H} auf $(L_{\text{infinite}})^C$

I. Für eine Eingabe $x \in \{0, 1, \#\}^*$ berechnet das Teilprogramm C, ob x die richtige Form hat(i.e. ob $x = \text{Kod}(M)\#w$ für eine TM M).

II. Falls nicht, verwirft B die Eingabe x.

III. Ansonsten, konstruiert C eine Turingmaschine M', die Eingaben ignoriert und immer M auf w simuliert. Wir sehen, dass M' genau dann hält, wenn M auf w hält.

IV. Folglich hält M' entweder für jede Eingabe (M hält auf w) oder für keine (M hält nicht auf w).

V. Da A genau dann akzeptiert, wenn die Eingabe keine gültige Kodierung ist(ausgeschlossen, da C das herausfiltert) oder wenn die Eingabe $M(x) = \text{Kod}(M')$ und M' für mindestens eine Eingabe hält, akzeptiert A $M(x)$ genau dann, wenn $x = \text{Kod}(M)\#w$ die richtige Form hat und M auf w hält.

Folglich gilt

$$x \in L_{\text{H}} \iff M(x) \in (L_{\text{infinite}})^C$$

$$\implies L_{\text{H}} \leq_R (L_{\text{infinite}})^C$$

Also folgt die Aussage

$$(L_{\text{infinite}})^C \in \mathcal{L}_R \implies L_{\text{H}} \in \mathcal{L}_R$$

Da wir $L_{\text{H}} \notin \mathcal{L}_R$ (**Satz 5.8**), folgt per indirekter Implikation:

$$(L_{\text{infinite}})^C \notin \mathcal{L}_R$$

■

9.11 Aufgabe 1

Zeige

$$L_{\text{diag}} \leq_{\text{EE}} L_{\text{H}}^{\complement}$$

Zur Erinnerung:

$$L_{\text{diag}} = \{w_i \in (\Sigma_{\text{bool}})^* \mid M_i \text{ akzeptiert } w_i \text{ nicht}\}$$

$$\begin{aligned} L_{\text{H}}^{\complement} = &\{\text{Kod}(M)\#w \in \{0,1,\#\}^* \mid M \text{ hält nicht auf } w\} \\ &\cup \{x \in \{0,1,\#\}^* \mid x \text{ nicht von der Form Kod}(M)\#w\} \end{aligned}$$

Lösung 1 Wir beschreiben einen Algorithmus A, so dass

$$x \in L_{\text{diag}} \iff A(x) \in L_{\text{H}}^{\complement}$$

Eingabe: $x \in (\Sigma_{\text{bool}})^*$

1. Findet i so dass $x = w_i$
2. Generiert $\text{Kod}(M_i)$
3. Generiert $\text{Kod}(\overline{M}_i)$ mit folgenden Modifikationen zu $\text{Kod}(M_i)$

   - Transitionen nach q_{reject} werden in eine Endlosschleife umgeleitet.

4. Gibt $\text{Kod}(\overline{M}_i)\#w_i$ aus.

Case Distinction

I. $\mathbf{x \in L_{\text{diag}}}$

$$\begin{aligned} &\implies M_i \text{ akzeptiert } x = w_i \text{ nicht} \\ &\implies \overline{M}_i \text{ hält nicht auf } w_i \\ &\implies A(x) = \text{Kod}(\overline{M}_i)\#w_i \in L_{\text{H}}^{\complement} \end{aligned}$$

II. $\mathbf{x \notin L_{diag}}$

$$\begin{aligned} &\implies M_i \text{ akzeptiert } x = w_i \\ &\implies \overline{M}_i \text{ hält auf } w_i \\ &\implies A(x) = \text{Kod}(\overline{M}_i)\#w_i \notin L_{\text{H}}^{\complement} \end{aligned}$$

■

10 Satz von Rice

Spezialfall des Halteproblems

Wir definieren $L_{\text{H},\lambda} = \{\text{Kod}(M) \mid M \text{ hält auf } \lambda\}$.

Lemma 5.8

$$L_{\text{H},\lambda} \notin \mathcal{L}_{\text{R}}$$

Beweis:

Wir zeigen $L_{\text{H}} \leq_{\text{EE}} L_{\text{H},\lambda}$. Wir beschreiben einen Algorithmus B, so dass $x \in L_{\text{H}} \iff B(x) \in L_{\text{H},\lambda}$.

Für jede Eingabe arbeitet B wie folgt:

- Falls x von der falschen Form, dann $B(x) = M_{inf}$, wobei M_{inf} unabhängig von der Eingabe immer unendlich läuft.
- Sonst $x = \text{Kod}(M)\#w$: Dann $B(x) = M'$, wobei M' die Eingabe ignoriert und immer M auf w simuliert.

Wir sehen, dass M' genau dann auf λ hält, wenn $x \in L_{\text{H}}$.

Daraus folgt $x \in L_{\text{H}} \iff B(x) \in L_{\text{H},\lambda}$.

■

Satz von Rice

Satz 5.9

Jedes semantisch nichttriviale Entscheidungsproblem über Turingmaschinen ist unentscheidbar.

- 'über Turingmaschinen' $= L \subseteq \mathbf{KodTM}$.
- 'nichttrivial' $= \exists M_1 : \text{Kod}(M_1) \in L$ und $\exists M_2 : \text{Kod}(M_2) \notin L$
- 'semantisch' = Für A, B mit $L(A) = L(B)$ gilt $\text{Kod}(A) \in L \iff \text{Kod}(B) \in L$.

10.1 Beispielaufgabe: Satz von Rice

Wir definieren

$$L_{\text{all}} = \{\text{Kod}(M) \mid L(M) = (\Sigma_{\text{bool}})^*\}.$$

Zeige

$$L_{\text{all}} \notin \mathcal{L}_{\text{R}}.$$

10.2 Satz von Rice - Beweis

10.2.1 Prerequisites

Zur Erinnerung:

Semantisch nichttriviales Entscheidungsproblem über TMs

Das Entscheidungsproblem (Σ, L), bzw. die Sprache L muss folgendes erfüllen.

I. $L \subseteq \mathbf{KodTM}$

II. $\exists M_1$ so dass $\text{Kod}(M_1) \in L$(i.e. $L \neq \emptyset$)

III. $\exists M_2$ so dass $\text{Kod}(M_2) \notin L$(i.e. $L \neq \mathbf{KodTM}$)

IV. Für zwei TM A und B mit $L(A) = L(B)$ gilt

$$\text{Kod}(A) \in L \iff \text{Kod}(B) \in L$$

KodTM $\subseteq (\Sigma_{\text{bool}})^*$ ist die Menge aller Kodierungen von Turingmaschinen.

Wir brauchen

Lemma 5.8

$$L_{H,\lambda} \notin \mathcal{L}_{\text{R}}$$

Zur Erinnerung:

$$L_{H,\lambda} = \{\text{Kod}(M) \mid M \text{ hält auf } \lambda\}$$

10.2.2 Idee

Wir zeigen für jedes **semantisch nichtriviale Entscheidungsproblem** (Σ, L)

$$L \in \mathcal{L}_{\text{R}} \implies L_{H,\lambda} \in \mathcal{L}_{\text{R}}$$

Aus dem folgt dann per Kontraposition

$$L_{H,\lambda} \notin \mathcal{L}_{\text{R}} \implies L \notin \mathcal{L}_{\text{R}}$$

Mit der Aussage $L_{H,\lambda} \notin \mathcal{L}_{\text{R}}$ von **Lemma 5.8**, können wir dann

$$L \notin \mathcal{L}_{\text{R}}$$

wie gewünscht folgern.

Wir müssen dann nur noch die Implikation

$$L \in \mathcal{L}_{\text{R}} \implies L_{H,\lambda} \in \mathcal{L}_{\text{R}}$$

beweisen.

Kernidee

Wir zeigen die **Existenz** einer Reduktion, aus der die Implikation folgt.

Konkret machen wir eine **Case Distinction** und zeigen jeweils

- Die **Existenz** einer EE-Reduktion von $L_{H,\lambda}$ auf L
  Daraus folgt $L_{H,\lambda} \leq_{\text{EE}} L$.
- oder die **Existenz** einer EE-Reduktion $L_{H,\lambda}$ auf $L^{\complement}$
  Daraus folgt $L_{H,\lambda} \leq_{\text{EE}} L^{\complement}$.

Zur Erinnerung:

Lemma 5.3

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen.

$$L_1 \leq_{\text{EE}} L_2 \implies L_1 \leq_{\text{R}} L_2$$

Weshalb reicht es $L_{H,\lambda} \leq_{\text{EE}} L^{\complement}$ zu zeigen?

Lemma 5.4

Sei Σ ein Alphabet. Für jede Sprache $L \subseteq \Sigma^*$ gilt:

$$L \leq_{\text{R}} L^{\complement} \text{ und } L^{\complement} \leq_{\text{R}} L$$

In beiden Cases folgt mit **Lemma 5.3** und **Lemma 5.4**, die gewünschte Aussage $L_{H,\lambda} \leq_{\text{R}} L$.

Explizit gilt nun

1. $$L_{H,\lambda} \leq_{\text{EE}} L^{\complement} \xRightarrow{\textbf{Lemma 5.3}} L_{H,\lambda} \leq_{\text{R}} L^{\complement} \xRightarrow{\textbf{Lemma 5.4}} L_{H,\lambda} \leq_{\text{R}} L$$
2. $$L_{H,\lambda} \leq_{\text{EE}} L \xRightarrow{\textbf{Lemma 5.3}} L_{H,\lambda} \leq_{\text{R}} L$$

Aus $L_{H,\lambda} \leq_{\text{R}} L$ folgt (in beiden Cases) die gewünschte Implikation

$$L \in \mathcal{L}_{\text{R}} \implies L_{H,\lambda} \in \mathcal{L}_{\text{R}}$$

10.2.3 Beweis

Sei $M_\emptyset$ eine TM s.d. $L(M_\emptyset) = \emptyset$.

Case Distinction

I. $\mathbf{Kod(M_\emptyset) \in L}$
   Wir zeigen $L_{H,\lambda} \leq_{\text{EE}} L^{\complement}$.
II. $\mathbf{Kod(M_\emptyset) \notin L}$
   Wir zeigen $L_{H,\lambda} \leq_{\text{EE}} L$.

Case I. $\mathbf{Kod(M_\emptyset) \in L}$

Es **existiert** eine TM $\overline{M}$, so dass $\text{Kod}(\overline{M}) \notin L$. (Nichttrivialität)

Wir beschreiben eine TM S, so dass für eine Eingabe $x \in (\Sigma_{\text{bool}})^*$

$$x \in L_{H,\lambda} \iff S(x) \in L^{\complement}$$

Daraus folgt dann die gewünschte EE-Reduktion.

Wir verwenden dabei $M_\emptyset$ und $\overline{M}$, da $\text{Kod}(M_\emptyset) \notin L^\complement$ und $\text{Kod}(\overline{M}) \in L^\complement$.

Case I. $\text{Kod}(M_\emptyset) \in$ L - Beschreibung von S

Eingabe $x \in (\Sigma_{\text{bool}})^*$

1. S überprüft ob $x = \text{Kod}(M)$ für eine TM M.
   Falls dies **nicht** der Fall ist, gilt $S(x) = \text{Kod}(M_\emptyset)$
2. Sonst $x = \text{Kod}(M)$. Dann $S(x) = \text{Kod}(A)$, wobei A wie folgt kodiert ist.
   i. Gleiches Eingabealphabet wie $\overline{M}$, i.e. $\Sigma_A = \Sigma_{\overline{M}}$.
   ii. Für eine beliebige Eingabe $y \in (\Sigma_{\overline{M}})^*$, simuliert A zuerst M auf λ **ohne die Eingabe** y **zu überschreiben**.
   iii. Danach simuliert A die TM $\overline{M}$ auf die gegebene Eingabe y.
   iv. Akzeptiert y genau dann, wenn $\overline{M}$ y akzeptiert.

Korrektheit

Wir zeigen

$$x \in L_{H,\lambda} \iff S(x) \in L^\complement$$

$(\implies)$:

Wir nehmen $x \in L_{H,\lambda}$ an und zeigen $S(x) \in L^\complement$.

Da M auf λ hält, wird A immer $\overline{M}$ auf der Eingabe y simulieren und wir haben $L(A) = L(\overline{M})$.

Da L (und somit auch $L^\complement$) ein **semantisches** Entscheidungsproblem ist, gilt

$$\text{Kod}(\overline{M}) \in L^\complement \implies \text{Kod}(A) \in L^\complement$$

Da die LHS der Implikation gegeben ist, folgt $S(x) = \text{Kod}(A) \in L^\complement$

$(\impliedby)$:

Wir nehmen $x \notin L_{H,\lambda}$ an und zeigen $S(x) \notin L^\complement$.

Aus Kontraposition folgt dann die gewünschte Rückimplikation.

Da M nicht auf λ hält, wird A bei jeder Eingabe nicht halten.

Somit folgt $L(A) = L(M_\emptyset)$ und da $\text{Kod}(M_\emptyset) \notin L^\complement$ per semantische Eigenschaft von L

$$S(x) = \text{Kod}(A) \notin L^\complement$$

Case II.

Zweite Case funktioniert genau gleich.

Wir haben $\text{Kod}(M_\emptyset) \notin L$.

Per Nichttrivialität existiert eine TM $\overline{M}$ mit $\text{Kod}(\overline{M}) \in L$.

...

■

11 EE Reduktion angewendet für $\mathcal{L}_{\text{RE}}$

11.1 Lemma zu RE-Reduktion

EE-Reduktion impliziert RE-Reduktion (**nicht in der Vorlesung**)

$$L_1 \leq_{\text{EE}} L_2 \implies (L_2 \in \mathcal{L}_{\text{RE}} \implies L_1 \in \mathcal{L}_{\text{RE}})$$

Beweis

Sei $L_1 \leq_{\text{EE}} L_2$ und $L_2 \in \mathcal{L}_{\text{RE}}$.

Wir zeigen nun $L_1 \in \mathcal{L}_{\text{RE}}$.

Per Definition von $L_1 \leq_{\text{EE}} L_2$ existiert ein Algorithmus F, der die Funktion $f : \Sigma_1^* \to \Sigma_2^*$ berechnet, so dass

$$\forall x \in \Sigma_1^*. x \in L_1 \iff f(x) \in L_2$$

Da $L_2 \in \mathcal{L}_{\text{RE}}$ existiert eine TM M_2 (die nicht unbedingt immer terminiert) mit $L(M_2) = L_2$.

Wir beschreiben mit F und M_2 nun eine TM M_1 mit $L(M_1) = L_1$.

Eingabe: $x \in \Sigma_1^*$

1. F berechnet auf x und übergibt seine Ausgabe $f(x)$ zur TM M_2
2. M_2 berechnet auf $f(x)$ und die Ausgabe wird übernommen.

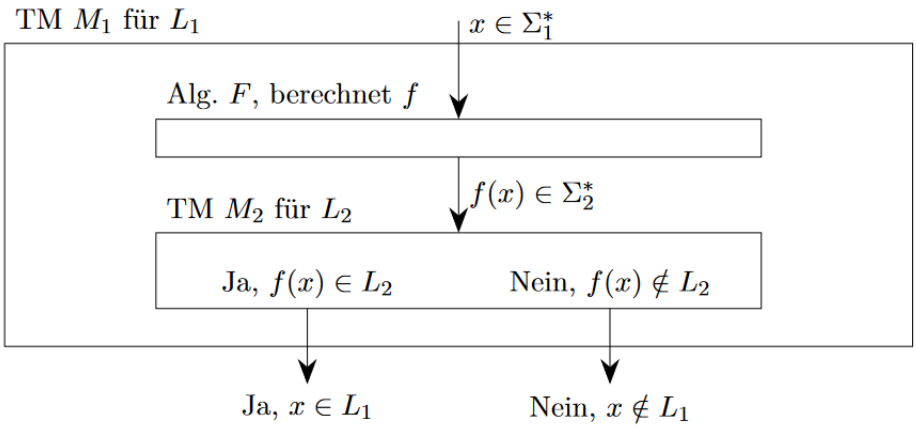


Figure 11: TM M_1, Zsf. Fabian Frei

Korrektheit $(L_1 = L(M_1))$

Case Distinction

I. $\mathbf{x \in L_1}$

$\implies f(x) \in L_2$ (Algorithmus F terminiert immer)

$L(M_2) = L_2 \implies f(x) \in L(M_2)$

da die Ausgabe von M_2 übernommen wird

$\implies x \in L(M_1)$

II. $\mathbf{x \notin L_1}$

$\implies f(x) \notin L_2$

$\implies f(x) \notin L(M_2)$

$\implies x \notin L(M_1)$

■

11.2 Verhältnis zwischen RE 'Reduktion' und R-Reduktion

$$L_1 \leq_{\mathrm{R}} L_2 \not\Longrightarrow (L_2 \in \mathcal{L}_{\mathrm{RE}} \implies L_1 \in \mathcal{L}_{\mathrm{RE}})$$

Wir beweisen diese Aussage per Gegenbeispiel.

Sei $L_1 = L_{\mathrm{diag}}$ und $L_2 = L_{\mathrm{diag}}^{\complement}$.

Wir haben

- $L_1 = L_{\mathrm{diag}} \notin \mathcal{L}_{\mathrm{RE}}$ (Satz 5.5)
- $L_2 = L_{\mathrm{diag}}^{\complement} \in \mathcal{L}_{\mathrm{RE}} \setminus \mathcal{L}_{\mathrm{R}}$ (Korollar 5.2, Lemma 5.5)

Per **Lemma 5.4** gilt $L_{\mathrm{diag}} \leq_{\mathrm{R}} L_{\mathrm{diag}}^{\complement}$.

Die rechte Implikation gilt jedoch nicht.

■

$$L_1 \leq_{\mathrm{R}} L_2 \not\Longleftarrow (L_2 \in \mathcal{L}_{\mathrm{RE}} \implies L_1 \in \mathcal{L}_{\mathrm{RE}})$$

Sei $L_1 = L_{\mathrm{U}}$ und $L_2 = \{0^i \mid i \in \mathbb{N}\}$.

Wir haben

- $L_1 = L_{\mathrm{diag}} \in \mathcal{L}_{\mathrm{RE}} \setminus \mathcal{L}_{\mathrm{R}}$ (Satz 5.6 und 5.7)
- $L_2 = \{0^i \mid i \in \mathbb{N}\} \in \mathcal{L}_{\mathrm{R}}$ (da $\mathcal{L}_{\mathrm{EA}} \subset \mathcal{L}_{\mathrm{R}}$)

Da $L_1 \in \mathcal{L}_{\mathrm{RE}}$, gilt die Implikation auf der rechten Seite für dieses L_1 und L_2.

Da per Definition

$$L_1 \leq_{\mathrm{R}} L_2 \iff (L_2 \in \mathcal{L}_{\mathrm{R}} \implies L_1 \in \mathcal{L}_{\mathrm{R}})$$

folgt aus $L_1 \notin \mathcal{L}_{\mathrm{R}}$ und $L_2 \in \mathcal{L}_{\mathrm{R}}$, dass diese Instanzierung von L_1 und L_2 ein Gegenbeispiel ist.

■

Relation zu EE-Reduktion

Wir haben aber gezeigt, dass

$$L_1 \leq_{\mathrm{EE}} L_2 \implies L_1 \leq_{\mathrm{R}} L_2$$

und

$$L_1 \leq_{\mathrm{EE}} L_2 \implies (L_2 \in \mathcal{L}_{\mathrm{RE}} \implies L_1 \in \mathcal{L}_{\mathrm{RE}})$$

Die Rückrichtung gilt jeweils nicht.

12 How To Reduktion

12.1 $L \in \mathcal{L}_{\mathbf{R}}$

Wir kennen zwei Methoden um dies zu beweisen:

I. **Reduktion**

(a) Wir finden eine Sprache $L' \in \mathcal{L}_{\mathrm{R}}$ (entweder schon in Vorlesung bewiesen oder selbst beweisen).

(b) Zeige die Reduktion $L \leq_{\mathrm{R}} L'$ (folgt trivial aus Lemma 5.4 für $L' = L^{\complement}$).

II. **Direkter Beweis: TM Konstruktion**

(a) Beschreibung einer TM (bzw. ein Algorithmus) M mit $L(M) = L$. Dabei kann man eine schon bekannte TM A verwenden, die immer hält (i.e. $L(A) \in \mathcal{L}_{\mathrm{R}}$).

(b) Beweise $L(M) = L$ und dass die TM M **immer** hält.

12.2 $L \notin \mathcal{L}_{\mathbf{R}}$

Wir kennen hier auch 3 Arten:

- **Trivial**

  Folgt sofort aus $L \notin \mathcal{L}_{\mathrm{RE}}$, da $\mathcal{L}_{\mathrm{R}} \subset \mathcal{L}_{\mathrm{RE}}$.

- **Reduktion**

  (a) Finde eine Sprache L', so dass $L' \notin \mathcal{L}_{\mathrm{R}}$ (muss bewiesen werden, falls nicht im Buch).

  (b) Beweise $L' \leq_{\mathrm{R/EE}} L$.

  (c) Geeignete Sprachen als L' sind: $L^{\complement}_{empty}, L^{\complement}_{diag}, L_{\mathrm{H}}, L_{\mathrm{U}}, L_{\mathrm{H},\lambda}$. (Alle im Buch bewiesen)

- **Satz von Rice**

12.3 Anwendung von Satz von Rice

Für den **Satz von Rice**:

- Wir können mit diesem Satz nur $L \notin \mathcal{L}_{\mathrm{R}}$ beweisen!
- Wir haben folgende Bedingungen:

  i. $L \subseteq$ KodTM

  ii. $\exists$ TM M: $\mathrm{Kod}(M) \in L$

  iii. $\exists$ TM M: $\mathrm{Kod}(M) \notin L$

  iv. $\forall$ TM M_1, M_2: $L(M_1) = L(M_2) \implies (\mathrm{Kod}(M_1) \in L \iff \mathrm{Kod}(M_2) \in L)$

  Für den letzten Punkt (4) muss man überprüfen, ob in der Definition von $L = \{\mathrm{Kod}(M) \mid M \text{ ist TM und } ...\}$ überall nur $L(M)$ vorkommt und nirgends M direkt.

  Beziehungsweise reicht es, wenn man die Bedingung so umschreiben kann, dass sie nur noch durch $L(M)$ beschrieben ist.

12.4 $L \in \mathcal{L}_{\mathbf{RE}}$

I. Wir beschreiben eine TM M mit $L(M) = L$, die nicht immer halten muss.

II. Meistens muss die TM eine Eigenschaft, für alle möglichen Wörter prüfen.

III. Bsp: $L = \{\mathrm{Kod}(M_1) \mid \mathrm{Kod}(M_1) \in L^{\complement}_{\mathrm{H}}\}$: Wir gehen alle Wörter durch, um dasjenige zu finden, für das M_1 hält.

IV. Wir verwenden oft einen von den folgenden 2 Tricks, um dies zu tun:

- Da es für jede NTM M', eine TM M gibt, so dass $L(M') = L(M)$, können wir eine solche definieren, für die $L(M') = L$ gilt.
- Die andere Variante, ist die parallele Simulation von Wörtern, bei dem man das Diagonalisierungsverfahren aus dem Buch verwendet. (Bsp: Beweis $L_{\mathrm{empty}} \in \mathcal{L}_{\mathrm{RE}}$, S. 156 Buch)

12.5 $L \notin \mathcal{L}_{\mathbf{RE}}$

Hier haben wir 2 mögliche (offizielle) Methoden:

- Diagonalisierungsargument mit Widerspruch, wie beim Beweis von $L_{\text{diag}} \notin \mathcal{L}_{\text{RE}}$.
- Widerspruchsbeweis mit der Aussage $L \in \mathcal{L}_{\text{RE}} \wedge L^{\complement} \in \mathcal{L}_{\text{RE}} \implies L \in \mathcal{L}_{\text{R}}$ (Aufgabe 5.22, muss begründet werden!).

Inoffiziell könnten wir auch die EE-Reduktion verwenden, wird aber weder in der Vorlesung noch im Buch erwähnt.

12.6 EE- und R-Reduktionen: Tipps und Tricks

- Die vorgeschaltete TM A muss immer terminieren! I.e. sie muss ein Algorithmus sein.
- Die Eingabe sollte immer zuerst auf die Richtige Form überprüft werden!

  Auch im Korrektsheitsbeweis, sollte dieser Fall als erstes abgehandelt werden.
- EE-Reduktion: Für Korrektheit müssen wir immer $x \in L_1 \iff A(x) \in L_2$ beweisen.
- Wir verwenden meistens folgende 2 Tricks:

  i. Transitionen nach q_{accept} oder q_{reject} umleiten nach q_{reject}/q_{accept} oder einer **Endlosschleife**.

  ii. TM M' konstruieren, die ihre Eingabe ignoriert und immer dasselbe tut (z.B. eine TM dessen Kodierung gegeben ist, auf ein fixes Wort simuliern).
- Die Kodierung einer TM generieren, dessen Sprache gewisse Eigenschaften hat(z.B. sie akzeptiert alle Eingaben, läuft immer unendlich etc.)
- Bei $L_1 \leq_{\text{R/EE}} L_2$ nehmen wir einen Algorithmus A_{L_2} an mit $L(A_{L_2}) = L_2$.

  Wir können A_{L_2} auch in der Kodierung einer TM M' verwenden, die wir dann zu A_{L_2} übergeben!

Fortgeschrittener Trick

Aufgabe

Sei $L_{\text{all}} = \{\text{Kod}(M) \mid M \text{ akzeptiert jede Eingabe}\}$.

Zeigen Sie $L_{\text{H}}^{\complement} \leq_{\text{EE}} L_{\text{all}}$.

Kernidee

Für eine Eingabe $x = \text{Kod}(M)\#w$, generieren wir $\text{Kod}(A)$ einer TM A, die folgendes folgendes macht:

A :

Eingabe y

1. Berechnet $|y|$ Schritte von M auf w.
2. Falls danach die Berechnung nach $|y|$ noch nicht terminiert hat, akzeptiert A die Eingabe y.
3. Sonst verwirft A die Eingabe.

$$A \text{ akzeptiert jede Eingabe} \iff M \text{ läuft unendlich auf } w$$

Wir nutzen, dass L_{all} unendlich viele Wörter hat, um implizit jede mögliche endliche Berechnungslänge abzudecken und eine Endlosschleife zu erkennen.

Bemerkung: Implikationsbeweis für Reduktion

Wenn eine Reduktion verlangt wird, dann dürft ihr die Implikation nicht trivial per Implikationsaussage zeigen.

Gemeint damit ist folgender Ansatz.

$L_1 \leq_R L_2$ soll gezeigt werden.

Da per Definition

$$L_1 \leq_R L_2 \iff (L_2 \in \mathcal{L}_R \implies L_1 \in \mathcal{L}_R)$$

folgt die gewünschte Aussage per $L_2 \notin \mathcal{L}_R$ (oder $L_1 \in \mathcal{L}_R$).

Dieser Ansatz gibt an der Prüfung 0 Punkte.

13 Komplexitätstheorie

13.1 Konfiguration

Wir erinneren uns:

Konfiguration einer k-Band-TM

Die Konfiguration einer k-Band-TM sieht wie folgt aus

$$(q, w, i, u_1, i_1, u_2, i_2, ..., u_k, i_k) \in Q \times \Sigma^* \times \mathbb{N} \times (\Gamma^* \times \mathbb{N})^k$$

wobei

- q der Zustand der TM ist
- ¢w\$ der Inhalt des Eingabebandes, Lesekopf Eingabeband auf dem i-ten Feld
- für $j \in \{1, ..., k\}$ ist der Inhalt des j-ten Bandes ¢u_j␣␣␣ und $i_j \leq |u_j|$ die Position des Kopfs auf dem j-ten Band.

13.2 Time

Sei M eine MTM oder TM, die immer hält. Sei Σ das Eingabealphabet von M. Sei $x \in \Sigma^*$ und $D = C_1, C_2, ..., C_k$ die Berechnung von M auf x.
Die **Zeitkomplexität $\mathbf{Time_M(x)}$ der Berechnung von M auf x** ist definiert durch

$$\mathbf{Time_M(x)} = k - 1.$$

Die **Zeitkomplexität von M** ist die Funktion $\text{Time}_M : \mathbb{N} \to \mathbb{N}$, definiert durch

$$\mathbf{Time_M(n)} = \max\{\text{Time}_M(x) \mid x \in \Sigma^n\}.$$

13.3 Space

Sei $k \in \mathbb{N} \setminus \{0\}$. Sei M eine k-Band-TM, die immer hält.
Sei

$$C = (q, x, i, \alpha_1, i_1, \alpha_2, i_2, ..., \alpha_k, i_k)$$
$$\text{mit } 0 \leq i \leq |x| + 1 \text{ und } 0 \leq i_j \leq |\alpha_j| \text{ für } j = 1, ..., k$$

eine Konfiguration von M.
Die **Speicherplatzkomplexität von C** ist

$$\mathbf{Space_M(C)} = \max\{|\alpha_i| \mid i = 1, ..., k\}.$$

Sei $C_1, C_2, ..., C_l$ die Berechnung von M auf x. Die **Speicherplatzkomplexität von M auf x** ist

$$\mathbf{Space_M(x)} = \max\{\text{Space}_M(C_i) \mid i = 1, ..., l\}.$$

Die **Speicherplatzkomplexität von M** ist die Funktion $\text{Space}_M : \mathbb{N} \to \mathbb{N}$, definiert durch

$$\mathbf{Space_M(n)} = \max\{\text{Space}_M(x) \mid x \in \Sigma^n\}.$$

Bemerkungen

1. Länge des Eingabewortes, hat keinen Einfluss auf die Speicherplatzkomplexität.
2. Mächtigkeit des Alphabets hat keinen Einfluss auf die Speicherplatzkomplexität.

Lemma 6.1

Sei $k \in \mathbb{N} \setminus \{0\}$. Für jede k-Band-TM A, die immer hält, existiert eine äquivalente 1-Band-TM B, so dass

$$\text{Space}_B(n) \leq \text{Space}_A(n)$$

Beweisskizze:

Gleiche Konstruktion wie in Lemma 4.2.

Lemma 4.2 = "Für jede MTM A existiert eine äquivalente TM B".

Wir sehen, dass B genau so viele Felder braucht, wie A.

Lemma 6.2

Zu jeder MTM A existiert eine äquivalente MTM B mit

$$\text{Space}_B(n) \leq \frac{\text{Space}_A(n)}{2} + 2$$

Beweisskizze:

Wir fassen jeweils 2 Felder von A zu einem Feld in B zusammen. $\Gamma_B = \Gamma_A \times \Gamma_A$. Wir addieren 1 für das ¢ am linken Rand und 1 für das Aufrunden im Fall von ungerader Länge.

13.4 Asymptotik

- $\mathcal{O}(\mathbf{f(n)})$:
  Menge aller Funktionen, die asymptotisch nicht schneller wachsen als $f(n)$.

- **$\Omega(g(n))$**:

  Menge aller Funktionen, die asymptotisch mind. so schnell wachsen wie $g(n)$.
- **$\Theta(h(n))$**:

  Menge aller Funktionen, die asymptotisch gleich schnell wachsen wie $h(n)$.

Small o-notation

Seien f und g zwei Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$.
Falls $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dann sagen wir, dass g asymptotisch **schneller wächst** als f:

$$f(n) \in o(g(n))$$

Bloomsches Speedup Theorem

Satz 6.1

Es **existiert** ein Entscheidungsproblem $(\Sigma_{\text{bool}}, L)$, so dass für jede MTM A, die $(\Sigma_{\text{bool}}, L)$ entscheidet, eine MTM B existiert, die auch $(\Sigma_{\text{bool}}, L)$ entscheidet, und für die gilt

$$\text{Time}_B(n) \leq \log_2(\text{Time}_A(n))$$

für unendlich viele $n \in \mathbb{N}$.

I.e. es existieren Entscheidungsprobleme, die keinen optimalen Algorithmus haben.

Deswegen fokussieren wir uns auf untere und obere Schranken der Komplexität eines Problemes und nicht auf die genaue Bestimmung davon.

Komplexität eines Entscheidungsproblems (Σ, L)

Sei L eine Sprache. Sei $f, g : \mathbb{N} \to \mathbb{R}^+$.

- $\mathcal{O}(g(n))$ ist eine **obere Schranke für die Zeitkomplexität von** L, falls eine MTM A **existiert**, die L entscheidet und $\text{Time}_A(n) \in \mathcal{O}(g(n))$.
- $\Omega(f(n))$ ist eine **untere Schranke für die Zeitkomplexität von** L, falls für **jede** MTM B die L entscheidet und $\text{Time}_B(n) \in \Omega(f(n))$.
- Eine MTM C heisst **optimal für** L, falls $\text{Time}_C(n) \in \mathcal{O}(f(n))$ und $\Omega(f(n))$ eine untere Schranke für die Zeitkomplexität ist.

Untere Schranke finden und beweisen: **schwierig**.

Obere Schranke kann durch einen konkreten Algorithmus gezeigt werden.

13.5 Komplexitätsklassen

Klassen

Für alle Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ definieren wir

$$\begin{aligned}
\mathbf{TIME(f)} &= \{L(B) \mid B \text{ ist eine MTM mit } \text{Time}_B(n) \in \mathcal{O}(f(n))\} \\
\mathbf{SPACE(g)} &= \{L(A) \mid A \text{ ist eine MTM mit } \text{Space}_A(n) \in \mathcal{O}(g(n))\} \\
\mathbf{DLOG} &= \text{SPACE}(\log_2 n) \\
\mathbf{P} &= \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c) \\
\mathbf{PSPACE} &= \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c) \\
\mathbf{EXPTIME} &= \bigcup_{d \in \mathbb{N}} \text{TIME}(2^{nd})
\end{aligned}$$

Zeitkomplexität zu Platzkomplexität

Lemma 6.3

Für jede Funktion $t : \mathbb{N} \to \mathbb{R}^+$ gilt

$$\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$$

Beweisskizze:

In $\mathcal{O}(t(n))$ Schritten sind höchstens $\mathcal{O}(t(n))$ Felder beschreibbar.

Korollar 6.1

$$\text{P} \subseteq \text{PSPACE}$$

13.6 Platz- & Zeitkonstruierbarkeit

EIne Funktion: $s : \mathbb{N} \to \mathbb{N}$ heisst **platzkonstruierbar**, falls eine 1-Band-TM M existiert, so dass

(i) $\text{Space}_M(n) \leq s(n)$ für alle $n \in \mathbb{N}$ und

(ii) für jede Eingabe 0^n, generiert M das Wort $0^{s(n)}$ auf ihrem Arbeitsband und hält in q_{accept}.

EIne Funktion: $t : \mathbb{N} \to \mathbb{N}$ heisst **zeitkonstruierbar**, falls eine MTM A existiert, so dass

(i) $\text{Time}_A(n) \leq t(n)$ für alle $n \in \mathbb{N}$ und

(ii) für jede Eingabe 0^n, generiert A das Wort $0^{t(n)}$ auf dem ersten Arbeitsband und hält in q_{accept}.

Platzgarantien

Lemma 6.4 (verständlicher formuliert)

Sei $s : \mathbb{N} \to \mathbb{N}$ platzkonstruierbar.
Für jede MTM M, für welche $\text{Space}_M(w) \leq s(|w|)$ nur für alle $w \in L(M)$ erfüllt, existiert eine äquivalente MTM A, welche dies für alle $w \in \Sigma^*$ erfüllt.

Beweisskizze:

Erzeuge für jede Eingabe $x \in \Sigma^*$ zuerst $0^{s(|x|)}$ auf einem zusätzlichen Band und nutze das als Platzüberwachung.

Wenn A diesen Platz überschreiten will, wird die Simulation unterbrochen und die Eingabe verworfen.

Zeitgarantien

Lemma 6.5 (verständlicher formuliert)

Sei $t : \mathbb{N} \to \mathbb{N}$ zeitkonstruierbar.
Zu jeder MTM M, welche $\text{Time}_M(w) \leq t(|w|)$ nur für alle $w \in L(M)$ erfüllt,
existiert eine äquivalente MTM A, welche zumindest $\text{Time}_A(w) \leq 2t(|w|) \in \mathcal{O}(t(|w|))$ für alle $w \in \Sigma^*$ erfüllt.

$$\implies \text{Time}_A(n) \in \mathcal{O}(t(n))$$

Beweisskizze:

Schreibe für jede Eingabe $x \in \Sigma^*$ $0^{t(|x|)}$ auf ein zusätzliches Arbeitsband und nutze dies zur Zeitzählung.

Wenn A mehr Schritte machen will, wird die Simulation abgebrochen und die Eingabe verworfen.

Speicherplatzkomplexität zu Zeitkomplexität

Satz 6.2

Für jede Funktion s mit $s(n) \geq \log_2(n)$ gilt:

$$\textbf{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \textbf{TIME}(c^{s(n)})$$

Beweis

Sei $L \in \textbf{SPACE}(s(n))$. Nach Lemma 6.1 existiert eine 1-Band-TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, die **immer hält**, so dass $L = L(M)$ und $\text{Space}_M(n) \leq d \cdot s(n)$ für $d \in \mathbb{N}$ gelten.

Für jede Konfiguration $C = (q, w, i, x, j)$ von M definieren wir die **innere Konfiguration von** C als

$$\text{In}(C) = (q, i, x, j).$$

Die innere Konfiguration enthält das Eingabewort w nicht, da dies sich während einer Berechnung nicht ändert.

Sei $\text{InKonf}_M(n)$ die Menge aller möglichen inneren Konfigurationen auf Eingabewörtern der Länge n.

Sei $X = |\text{InKonf}_M(n)|$ dessen Kardinalität.

Sei $D = C_1C_2...C_k$ eine endliche Berechnung von M auf einem Wort $w, |w| = n$.

Wir zeigen per Widerspruch, dass D maximal X verschiedene Konfigurationen haben kann, i.e. $k \leq X$.

Nehmen wir zum Widerspruch an $k > X$.

Dann muss es in $D = C_1C_2...C_i...C_j...C_k$, zwei identische innere Konfigurationen $\text{In}(C_i)$ und $\text{In}(C_j)$ geben (für $i < j$).

Da M deterministisch ist, sollte aber von $C_i = C_j$ aus immer die gleichen Berechnungsschritte ausgeführt werden.

Dann wäre aber D eine unendliche Berechnung mit der Endlosschleife $C_iC_{i+1}...C_j$. **Widerspruch**, da M immer hält.

Eine beliebige endliche Berechnung D von M auf $w, |w| = n$, kann höchstens X viele Zeitschritte (i.e. Konfigurationen) haben.

Jetzt müssen wir noch $X = |\text{InKonf}_M(n)|$ abschätzen.

Wir wissen folgendes

- Es gibt $|Q|$ verschieden mögliche Zustände.
- Index des Eingabekopfes ist $0 \leq i \leq n+1$ (Eingabeband ¢w\$ mit $|w| = n$)
- Inhalt des Arbeitsbandes x hat Länge: $|x| \leq \text{Space}_M(n) \leq d\dot{s}(n)$
- Index vom Kopf auf dem Arbeitsband: $0 \leq j \leq \text{Space}_M(n) \leq d \cdot s(n)$
- $x \in \Gamma^{|x|}$
- $n + 2 \leq 4^{\log_2 n} \leq 4^{s(n)}$ für $n \geq 2$

Setzen wir alles zusammen:

$$\begin{aligned} |\text{InKonf}_M(n)| &\leq |Q| \cdot (n+2) \cdot |\Gamma|^{\text{Space}_M(n)} \cdot \text{Space}_M(n) \\ &\leq (\max\{4, |Q|, |\Gamma|\})^{4d \cdot s(n)} \\ &\leq c^{s(n)} \end{aligned}$$

■

14 NP-Vollständigkeit

14.1 Verifikation

Verfizierer

Sei $L \subseteq \Sigma^*$ eine Sprache und sei $p : \mathbb{N} \to \mathbb{N}$ eine Funktion.
Ein Algorithmus A (MTM) ist ein **p-Verfizierer für L** mit $\mathbf{V(A) = L}$, falls A mit folgenden Eigenschaften auf allen Eingaben aus $\Sigma^* \times (\Sigma_{\text{bool}})^*$ arbeitet:

(i) $\text{Time}_A(w, x) \leq p(|w|)$ für jede Eingabe $(w, x) \in \Sigma^* \times (\Sigma_{\text{bool}})^*$.

(ii) Für jedes $w \in L$ existiert ein $x \in (\Sigma_{\text{bool}})^*$, so dass $|x| \leq p(|w|)$ und $(w, x) \in L(A)$. Das Wort x nennt man einen **Beweis** oder einen **Zeugen** der Behauptung $w \in L$.

(iii) Für jedes $y \notin L$ gilt $(y, z) \notin L(A)$ für alle $z \in (\Sigma_{\text{bool}})^*$.

Polynomialzeitverifikation

Falls $p(n) \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$, so sagen wir, dass A ein **Polynomialzeit-Verfizierer** ist.
Wir definieren die **Klasse der in Polynomialzeit verifzierbaren Sprachen** als

$$\mathbf{VP} = \{V(A) \mid A \text{ ist ein Polynomialzeit-Verfizierer}\}.$$

Satz 6.8

VP = NP

Die Klasse NP ist demnach die Klasse aller Sprachen L, die für jedes $x \in L$ einen in $|x|$ polynomiell langen Beweis von "$x \in L$" haben, welchen man deterministisch in polynomieller Zeit verifizieren bekann.

Dies können wir benutzen, um $L \in$ NP zu beweisen!

14.2 P-Reduktion

Seien $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Wir sagen, dass **L_1 polynomiell auf L_2 reduzierbar ist, $L_1 \leq_p L_2$**, falls eine polynomieller Algorithmus A existiert, der für jedes Wort $x \in \Sigma_1^*$ ein Wort $A(x) \in \Sigma_2^*$ berechnet, so dass

$$x \in L_1 \iff A(x) \in L_2$$

A wird eine **polynomielle Reduktion** von L_1 auf L_2 genannt.

Bemerkung

Analog zur EE-Reduktion, nur das A jetzt noch polynomiell laufen muss.

Begrifflichkeiten

Eine Sprache L ist **NP-schwer**, falls für alle Sprachen $L' \in$NP gilt $L' \leq_p L$.
Eine Sprache L ist **NP-vollständig**, falls

(i) $L \in$NP und

(ii) L ist NP-schwer.

Lemma 6.7

Falls $L \in P$ und L ist NP-schwer, dann gilt P = NP.

Satz von Cook

Wir haben

$$\text{SAT} = \{x \in (\Sigma_{\text{logic}})^* \mid x \text{ kodiert eine erfüllbare Formel in KNF}\}$$

Satz 6.9

SAT ist NP-vollständig.

Da

$$L_1 \leq_p L_2 \implies (L_2 \in \text{P} \implies L_1 \in \text{P})$$

können wir mit diesem Resultat die NP-Schwere anderer Probleme einfacher beweisen.

14.3 Klassische Probleme

$$\begin{aligned}
\text{SAT} &= \{\phi \mid \phi \text{ ist eine erfüllbare Formel in KNF}\} \\
\text{CLIQUE} &= \{(G,k) \mid G \text{ ist ein ungerichteter Graph, der eine } k\text{-Clique enthält}\} \\
\text{VC} &= \{(G,k) \mid G \text{ ist ein ungerichteter Graph mit einer Knotenüberdeckung} \\
&\qquad \text{(vertex cover) der Mächtigkeit höchstens } k\}
\end{aligned}$$

Higher-Level of Abstraction

Wir müssen uns nicht mehr überlegen, wie die Probleminstanzen als endliche Wörter kodiert sind!

Das heisst auch, dass ihr nicht mehr explizit auf falsche Form überprüfen müsst.

Ihr könnt annehmen, dass die Eingabe jeweils schon eine wohlgeformte Instanz des Problems ist.

14.4 Aufgabe 6.22.a

Beweise

$$\text{VC} \leq_p \text{CLIQUE}$$

Zur Erinnerung:

$$\begin{aligned}\text{CLIQUE} &= \{(G,k) \mid G \text{ ist ein ungerichteter Graph, der eine } k\text{-Clique enthält}\}\\ \text{VC} &= \{(G,k) \mid G \text{ ist ein ungerichteter Graph mit einer Knotenüberdeckung}\\ &\qquad \text{(vertex cover) der Mächtigkeit höchstens } k\}\end{aligned}$$

Ein VC ist eine Knotenmenge $C \subseteq V$, so dass

$$\forall \{u,v\} \in E. v \in C \vee u \in C.$$

Dies kann ein bisschen non-intuitiv sein, da die Knotenmenge eigentlich alle Kanten überdecken muss:)

Wir beschreiben einen polynomiellen Algorithmus A:

Eingabe $(G = (V,E), k)$ für VC

1. Findet $\overline{G} = (V, \overline{E})$ mit $\overline{E} = \{\{u,v\} \mid u,v \in V, u \neq v, \{u,v\} \notin E\}$
2. Gibt $(\overline{G}, |V| - k)$ aus.

Es sollte klar sein, dass A polynomiell läuft.

Korrektheit

Wir beweisen nun

$$S \subseteq V \text{ ist ein Vertex Cover von } G \iff V \setminus S \text{ ist eine Clique von } \overline{G}$$

$(\implies)$:

Sei $S \subseteq V$ ein Vertex Cover von G.

$\implies$ Per Definition gilt für jede Kante $\{u,v\} \in E$ mindestens $u \in S$ oder $v \in S$.

$\implies$ Also existiert keine Kante $\{u,v\} \in E$ mit $u,v \in V \setminus S$.

$\implies$ Deshalb gilt für alle $u,v \in V \setminus S, u \neq v$, dass $\{u,v\} \in \overline{E}$.

$\implies$ $V \setminus S$ ist eine Clique in $\overline{G}$.

$(\impliedby)$:

Sei $V \setminus S$ eine Clique in $\overline{G}$.

$\implies$ Per Definition gilt für alle Knotenpaare $u,v \in V \setminus S, u \neq v$ jeweils $\{u,v\} \in \overline{E}$.

$\implies$ Also existiert keine Kante $\{u,v\} \in E$ mit $u,v \in V \setminus S$.

$\implies$ Deshalb gilt für alle $\{u,v\} \in E$, dass $u \in S$ oder $v \in S$.

$\implies$ S ist ein Vertex Cover in G.

Mit der Aussage

$$S \subseteq V \text{ ist ein Vertex Cover von } G \iff V \setminus S \text{ ist eine Clique von } \overline{G} \qquad (1)$$

können wir nun die Korrektheit beweisen.

$$\begin{aligned}(G,k) \in \text{VC} &\iff \exists S \subseteq V : S \text{ ist ein VC von } G \text{ und } |S| \leq k \\ &\iff V \setminus S \text{ ist eine Clique von } \overline{G} \text{ und } |V \setminus S| \geq |V| - k \\ &\iff (\overline{G}, |V| - k) \in \text{CLIQUE} \\ &\iff A((G,k)) \in \text{CLIQUE}\end{aligned}$$

■

15 How To P-Reduktion

15.1 PROBLEM ∈ NP

Beschreibung eine NTM M, die PROBLEM erkennt mit folgender Form:

1. M errät für eine Eingabe x nicht deterministisch ein Zertifikat/Beweis (z.B. eine erfüllende Belegung für SAT oder eine Clique für CLIQUE).
2. M verfiziert das Zertifikat deterministisch in Polynomialzeit.

Korrektheit

$x \in$ PROBLEM $\iff$ Es existiert ein solches Zertifikat $\iff$ Es existiert eine akzept. Berechnung von M auf x $\iff$ $x \in L(M)$

15.2 PROBLEM ist NP-schwer

Beweise

$$\text{OTHERPROBLEM} \leq_p \text{PROBLEM}$$

für ein anderes OTHERPROBLEM, dass NP-schwer ist (in der Vorlesung gezeigt oder ähnl.).

Alternativ könnte man auch etwas wie den Beweis vom Satz von Cook machen. **(Don't)**

15.3 PROBLEM ist NP-Vollständig

Zeige

1 PROBLEM ∈ NP

2 PROBLEM ist NP-schwer

15.4 OTHERPROBLEM $\leq_p$ PROBLEM

1. **Beschreibung** eines Algorithmus A, so dass

$$x \in \text{OTHERPROBLEM} \iff A(x) \in \text{PROBLEM}$$

2. **Korrektsheitsbeweis** von

$$x \in \text{OTHERPROBLEM} \iff A(x) \in \text{PROBLEM}$$

(nichttrivial)

3. **Polynomialzeit** von A beweisen/argumentieren. Meistens recht einfach.

15.5 Idee finden für Polynomialzeitreduktion

Grundsätzlich 2 Typen von Problemen/Sprachen:

1 Satisifiability: Logische Formeln (deren Erfüllbarkeit). Beispielsweise SAT und 3SAT, 4SAT...
2 Graphenprobleme: Eigenschaften von Graphen. Beispielsweise CLIQUE, VC, DS etc.

15.5.1 Satisfiablity zu Satisfiability - Idee

Meist müssen wir einzelne Klauseln umschreiben. Veränderung der Anzahl Literale.

- **Literale verringern.** (Für allg. siehe im Buch, Lemma 6.11)
  Beispiel: 6SAT $\leq_p$ 4SAT

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \mapsto (y_1 \vee x_1 \vee x_2 \vee x_3) \wedge (\overline{y}_1 \vee x_4 \vee x_5)$$

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \mapsto (y_1 \vee x_1 \vee x_2 \vee x_3) \wedge (\overline{y}_1 \vee x_4 \vee x_5 \vee x_6)$$

  Beispiel: E8SAT $\leq_p$ E4SAT

$$\begin{aligned}(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6 \vee x_7 \vee x_8) \mapsto &(y_1 \vee x_1 \vee x_2 \vee x_3) \wedge (\overline{y}_1 \vee x_4 \vee x_5 \vee y_2) \\ &\wedge (\overline{y}_2 \vee x_6 \vee x_7 \vee x_8)\end{aligned}$$

- **Mehr Literale.**
  Beispiel: 5SAT $\leq_p$ E5SAT

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \mapsto (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5)$$

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \mapsto (x_1 \vee x_2 \vee x_3 \vee x_4 \vee y_1) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4 \vee \overline{y}_1)$$

$$\begin{aligned}(x_1 \vee x_2 \vee x_3) \mapsto &(x_1 \vee x_2 \vee x_3 \vee y_1 \vee y_2) \\ &\wedge (x_1 \vee x_2 \vee x_3 \vee \overline{y}_1 \vee y_2) \\ &\wedge (x_1 \vee x_2 \vee x_3 \vee y_1 \vee \overline{y}_2) \\ &\wedge (x_1 \vee x_2 \vee x_3 \vee \overline{y}_1 \vee \overline{y}_2)\end{aligned}$$

  etc.
- **Mehr erfüllende Belegungen.**
  Füge Klausel hinzu. I.e.

$$\phi \mapsto \phi \wedge (y_1 \vee y_2)$$

  verdreifacht die Anzahl der erfüllenden Belegungen.

15.5.2 Satisfiability zu Satisfiability Reduktion - OTHERPROBLEM $\leq_p$ PROBLEM

Schema

Sei $F = F_1 \wedge ... \wedge F_m$ eine KNF Formel vom Typ OTHERPROBLEM

Eingabe für A: F

- A konstruiert $B = B_1 \wedge ... \wedge B_m$ mit einem der Tricks von oben.

Korrektheit

$$\begin{aligned} F \in \text{OTHERPROBLEM} &\iff F \text{ erfüllbar} \\ &\iff \text{Es existiert eine Belegung } \varphi : \varphi(F) = 1 \\ &\iff \exists \varphi : \varphi(F_i) = 1 \forall i \in \{1, ..., m\} \\ &\text{... Argumentation für beliebige Klausel} \\ &\iff \exists \varphi' : \varphi'(B_i) = 1 \forall i \in \{1, ..., m\} \\ &\iff B \text{ erfüllbar} \\ &\iff B \in \text{PROBLEM} \end{aligned}$$

15.5.3 Graphproblem zu Graphproblem - Reduktion

Patterns

1. Aus $G = (V, E)$ Komplementgraph $\overline{G}$ bilden. I.e. $\overline{G} = (V, \overline{E})$ mit
$$\overline{E} = \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$$
2. Eingabetupel (G, k) zu $(G, n - k)$ abbilden ($n = |V|$).
3. Beides davon (siehe CLIQUE zu VC).
4. Ersetzen einer Kante durch anderes Konstrukt
    - 2 Kanten mit Knoten dazwischen
    - Knoten hinzufügen für jede Kante und mit beiden Eckpunkten verbinden

15.5.4 Satisfiability zu Graphproblem

Beispiel: Lemma 6.9 SAT $\leq_p$ CLIQUE

Denkt über die Reduktion vom Satisfiability Problem zu SAT und von CLIQUE zum Graphproblem.

Versucht diese Abbildungen zu verknüpfen.

15.5.5 Graphenproblem zu Satisfiability

Vielleicht **Satz von Cook** verwenden?

Scheint sehr komplex eine konkrete Reduktion zu machen.

PROBLEM $\leq_p$ SAT

Generelle Idee

- Denke über Zertifikate für PROBLEM nach. Welche Bestandteile haben sie?

  Beispielsweise im Fall von Clique ist das Zertifikat eine Knotenmenge (alle untereinander verbunden).
- Kreiert eine Variablenmenge mit einer Variable pro mögliches Element für das Zertifikat.

  I.e. im Fall von CLIQUE eine Variable pro Knoten
- I.e. Wir werden dann für eine Belegung der Formel wissen welche Knoten im Zertifikat enthalten sind.
- Baue die Formel, die die Bedingungen kodiert, damit genau dann erfüllt sein kann, falls es ein Zertifikat gibt.